UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

A New Approach to Embedded Computer Benchmarking

WILLEM A SMIT

THESIS PRESENTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN ELECTRONIC ENGINEERING AT THE UNIVERSITY OF STELLENBOSCH

SUPERVISOR: PROF P.J. BAKKES

APRIL 2006

Declaration

I, the undersigned hereby declare that the work contained in this thesis is my own original work unless otherwise stated, and has not previously, in its entirety or in part, been submitted at any university for a degree.

28/02/2006

Date

Abstract

Years of experience gained in the field while designing and building new digital hardware for clients in both the private and public sectors have shown that choosing the correct processor for the job beforehand is difficult.

The reason is the type of situation where the final speed of the application cannot be known before it is tested on the actual hardware, but that the hardware cannot be built before it is known which CPU will be fast enough to run the application.

Designers have traditionally tried to reduce the risk by referring to various benchmark programs to compare processors with each other, and then by over designing. In this regard various attempts have been made to describe the performance of CPU's, but these are generally very application-specific and the accuracy depends on who is doing the measuring. Manufactures have been known to quote figures that will portray their hardware in the best possible light.

The purpose of this thesis will be to develop a robust, simple and quick way to determine what performance a CPU will achieve in a given practical application. The theory will then be tested on several CPU platforms.

The development of such a method has very practical application in the engineering industry, as the type and number of CPU's in a design have very real cost implications. The method will also have spin-offs in the System-On-a Chip (SOIC) and FPGA environment when the method is used to analyse the requirements of a given application. The results can then be used to influence the data flow paths and CPU architecture of such a design.

The thesis first does a literature survey of current benchmarking methods. This will then be used to influence the postulation of a theory of how the problem of benchmarking should be tackled. The theory will then be tested on several platforms, as stated above.

Opsomming

Verskeie jare se ervaring in die ontwerp en bou van digitale hardeware vir kliente in beide die privaat en publieke sektor het getoon dat dit nogal moeilik is om vooraf te besluit watter verwerker gebruik moet word.

Dit is die gevolg van 'n situasie waar daar nie geweet kan word hoe vinnig 'n gegewe verwerker 'n toepassing sal uitvoer nie alvorens die hardeware gebou en die toepassingsagteware op die uiteindelike hardeware getoets is nie.

Ontwerpers probeer tradisioneel die risiko's verminder deur verskillende verwerkers met mekaar vergelyk deur na enigeen van verskeie 'benchmark'-programme te verwys, en deur dan te oorontwerp. In hierdie verband is daar oor die laaste sewentig jaar verskeie pogings aangewend om verwerkerwerkverrigting op 'n standaardwyse te beskryf. Die probleem is dat hierdie pogings oor die algemeen baie toepassingspesifiek is, en dat die akkuraatheid afhang van wie die meetwerk doen. Vervaardigers is bekend daarvoor dat hulle slegs die syfers aanhaal wat hulle hardeware in die beste moontlike lig stel.

Die doel van hierdie tesis is om 'n vinnige maar robuuste en eenvoudige manier te ontwikkel waardeur die verrigting van 'n verwerker vir enige praktiese toepassing bepaal kan word. Die metode sal dan op 'n aantal verkers getoets word.

Die ontwikkeling van so 'n metode het verskeie praktiese toepassings deurdat die tipes en aantal verwerkers in 'n ontwerp groot koste-implikasies het. Die metode sal ook deur die SOIC- en FPGA-gemeenskap gebruik kan word deur die behoeftes van die toepassing te analiseer, en dan doelgemaakte hardeware te gaan bou wat daardie behoefte bevredig. Die resultate kan ook gebruik word om die datavloeipaaie en argitektuur van nuwe verwerkers te beïnvloed.

Hierdie tesis doen ten eerste 'n deeglike literatuuroorsig van huidige 'benchmark'-metodes. Dit word dan gebruik om te besluit hoe die probleem om 'n nuwe metode te ontwikkel, aangepak behoort te word. Hierdie teorie word dan op verskeie platforms getoets.

Acknowledgements

This project would not have been possible without the guidance and support of several people. They are my supervisor, Prof. Pieter Bakkes, without whose support this project would not have started, and my wife and family, who relocated over the length of the country to enable me to study, and at times at no small cost to themselves.

The author would further like to thank his colleagues both here at Stellenbosch University and at Kentron for their support. In this regard the author wishes to especially thank his supervisor at Kentron, Phillip Minnaar for his huge support, and Dr Thomas Niesler for asking the right questions and generally helping to keep him motivated.

Another great thanks goes to my father for his support, and especially for language-editing this document.

Willem Smit

December 2005

Contents

Acronyms and Abbreviations

3D	Three-dimensional
ALU	Arithmetic Logic Unit
CAN	Controller Area Network
CMYK	Cyan Magenta Yellow Black
COM	Communication port
CPU	Central Processor Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
ECL	EEMBC Certification Labs
EEMBC	EDN Embedded Microcomputer Benchmark Consortium
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
HINT	Hardware Integration
IDCT	Inverse Discrete Fourier Transform
IFFT	Inverse Fast Fourier Transform
I/O	Input Output
JPEG	Joint Photographic Experts Group
KWIPS	Kilo Whetstones Per Second
MFLOPS	Million Floating-Point Operations Per Second
MIPS	Million Instructions Per Second
MOPS	Million Operations Per Second
NOP	No Operation
NP	Non-linear Polynomial Time
PNG	Portable Network Graphics
PC	Personal Computer
QUIPS	Quality Improvement Per Second
RGB	Red Green Blue
RISC	Reduced Instruction Set Computer
SPARC	Scalable Processor Architecture
SHARC	Super Harvard Architecture

SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SPEC	Standard Performance Evaluation Corporation
VLIW	Very Long Instruction Word
YIQ	Luminance In-phase Quadrature

List of Figures

List of Tables

1. Scope

The subject of computer benchmarking has long been considered a difficult and complex problem. It can be regarded as NP Complex [1][2]. The history of benchmarking certainly seems to bear this out. Both academic institutions and commercial organisations have worked on the problem of a single killer benchmark application. The result almost always ends up as a set of programs which attempts to fully describe the performance envelope of the machine under test. These programs often do not adequately describe the complex interactions between the machine instruction set, its internal architecture and the external environment. This, in turn, leads to exploitation and confusion as commercial companies try to prove the supremacy of their hardware by tweaking the end-user perception.

This thesis is an attempt to provide a way of determining the processor performance envelope in such a way that it is not only more difficult to tweak the results but that the machine strengths and weaknesses can be surveyed at a glance.

Since the focus of the thesis is also on the end user, it should also be recognised that the end user compares CPU performance relative to his or her application. The benchmark should therefore also provide a way in which a CPU profile can be referenced to the user's application. It therefore touches on another holy grail of theoretical computing, namely that of performance prediction.

It is also recognised that a single performance figure as returned by most benchmarks cannot adequately represent the performance profile of a processor. This fact is also at the root of much quibbling amongst manufacturers and clients alike. This thesis will therefore endeavour to produce a 3D graph that will describe the performance envelope of the processor under test in such a way that the processor performance can be related to application performance on that processor.

This graph then should not only be easy and intuitive to set up, but it should be accurate to the extent that useful performance predictions can be derived from it. As such it should have the

ability to be a superset of other benchmarks. In other words, the benchmark should be able to predict the performance of other benchmarks as well.

The purpose of this benchmark is not to prove how wrong or right perceptions or marketing of a particular processor are. It is simply a tool to help the end user decide if the particular processor will be suited to his application or not. This graph should then also give an indication of how well the application field will suit the processor.

Benchmarks, however, are not the be-all and end-all of processor selection. In the commercial world (where most embedded applications are seated), once a minimum performance level is met, cost becomes the driving factor. And that cost can assume many forms such as:

- Silicon cost
- The familiarity of the available personnel with that architecture (man-hours needed to make a new architecture work)
- The availability of tools such as compilers and emulators and the cost of acquiring it
- Lifecycle expectancies. By this is meant that it is expensive to replace an architecture. The major cost component is not so much the actual hardware as the rewriting of the software for a new processor. In any given development project nothing is usually more expensive than the accumulated cost of the people you have to employ to do the engineering. Companies do not want to do that too often.

Being able to more accurately determine the suitability of an architecture for the task at hand should enable decision makers to more easily make the trade-off between cost and performance.

In the benchmarking environment the type of system under test is also important. Most of the research papers focus on parallel systems, and are intended for large computer systems. In the embedded world the focus is more on processor level. A second level of benchmarking will include the interface between the machine and its environment.

While this thesis will discuss benchmarks in general, and although the method proposed can be used on any CPU, the actual focus of the document is embedded computer benchmarks.

2. Survey of Current Benchmarking Methods and Software

2.1 Definition

So, what then is a benchmark? A tricky question, and one the author will attempt to answer as well as he can without painting himself into a corner.

Humans in general, and the male species in particular, tend to be concerned with measuring things. Am I stronger than my class mate, can I run faster than he, is my car faster, or was I in fact exceeding the speed limit when that camera caught me? How old is the world, and how far away is that galaxy? What is the time now, and how many milliseconds do I have spare from when the control algorithm updates the vectors in my inherently unstable flying machine until the machine attempts a manoeuvre which will destroy the airframe? How fast is the CPU, and will it be able to do the job required of it within the time limit mandated by the physics of the flying machine?

A processor benchmark, then, is simply the measuring of the performance of the said processor.

It should be noted that there are differences of opinion on what benchmarks are, and what a benchmark should measure. In the words of Dr Reinhold Weicker, in his original paper outlining the Dhrystone benchmark [3]:

"As a high-level language host, a computer architecture should execute efficiently those features of a programming language that are most frequently used in actual programs. This ability is often measured by a program known as a 'benchmark'"

This point also highlights the differences between the communities which practise benchmarking in that it is clear that some communities are interested in the native performance of a processor whereas others want to determine its ability as a high-level language host and still others are only interested in the performance of a set of applications on

the machine. It should be noted that many of the formal benchmarks originating from the academic community are written in a high-level language, and that the efficiency of the compiler is something they want to include in the benchmark score.

At this point the author would like to add a definition of his own, that of the embedded processor benchmark:

An embedded processor benchmark is a method of measuring the native performance (i.e. without a high-level compiler) of an embedded processor, in such a way that its suitability for the task at hand can be determined.

The implication is that the intended application should be completely known, and secondly that the benchmark should characterise the native, raw performance of the machine over a range of conditions in such a way that the results will tell that, all things considered, a means of programming could be found that will make the machine meet the demands of the application or not. The rationale is that the embedded world in its nature is either a high-volume application (commercial world) or a technically complex one (military world) or both (industrial applications). In most of these cases the business model will demand that manpower be expended in optimising the software (including assembly optimisations) to meet the demands of the application. In the commercial world the demand will be to make the unit cost be as low as possible, and that will mean using the lowest cost technology (and therefore cheapest processor) that the application can tolerate. There are many 8051 applications out there which were written only in assembly language by some of the brightest programmers the author have met in his career. In the industrial and military world, especially insofar machine vision applications are concerned, the fact of the matter is that the state of technology is such that the slowdown caused by a high-level complier (6 to 10 times [4]) will mean that either dedicated hardware will be built, or that the high bandwidth portions of the code will be written in assembler. And any project manager will tell you that if you have to choose between programming a processor (even in assembly) or designing a custom FPGA, writing the code on the processor is always cheaper.

Now on to the issue of which yardsticks (or metrics, as some prefer to name it) to use.

The knotty thing about measuring anything is choosing the yardstick to use. On the issue of the age of creation, the creationist and the evolutionists have fun times arguing the yardstick. So too with processors. The benchmark in essence is the yardstick, but not all agree on the form the yardstick should take. As such benchmark units and methodologies have proliferated. Also, different yardsticks tend to be used by those who use the machines and those who manufacture them.

2.2 Types of Benchmarks

Most current benchmarks use time as their point of reference, measured either in cycles or in elapsed time.

Three types of benchmark can generally be distinguished [5]. They are:

2.2.1 Synthetic Benchmarks

Synthetic benchmarks are based on synthetic code. With this is meant code that has no relationship to any real-world application, but are written to test specific areas of a processor's performance envelope.

Benchmarks in this class range from the public favourite and often misused MIPS (Million Instructions Per Second) rating, include the equally misused SpecInt and SpecF benchmarks, and stretch to more modern variants such as Dhrystone and Linpak.

Synthetic benchmarks are very prevalent and almost all modern-day performance figures quoted on processor data sheets are those of synthetic benchmarks. The Linpak and Dhrystone benchmarks are also prevalent throughout the workstation and parallel computing communities.

2.2.2 Real-world benchmarks.

These benchmarks are based on real-world applications. They are often used in the PC and Microsoft Windows environment and are usually application-based around applications such as Microsoft Word, Excel and Paintshop Pro. The benchmark process usually includes measuring the execution time of some standard operation in the specific application.

2.2.3 Hybrids.

These are synthetic benchmarks that are based around real-world applications. It is more often used by the embedded computing community. Examples are the FFT (Fast Fourier Transform) execution times used by vendors of DSP processors and the set of benchmarks used by the EEBMC forum.

As can be seen form the above descriptions, the application fields that use these benchmarks are also quite different from each other. In general, they are the high-end scientific computing community (this includes the parallel processing and workstation communities), the desktop community (PC's etc) and the embedded processor community.

It is interesting to note that most of the research around computer benchmarks is done by the scientific computing community, and that many of the synthetic benchmarks of today have their origins within that community. Very little research has been done into embedded computer benchmarks, and those that do exist are either derived form the scientific community or are of a commercial nature (the EEBMC comes to mind). The paradox, however, is that the embedded processor market at $3 billion a year far outstrips that of all the other processors sold several times over. The venerable Pentium processor, which has cornered 95% of the desktop PC market, only makes up 2% of total processor sales. The question, then, is how much money can be saved by the embedded processor community if they have an accurate way of determining the best processor for their application by ensuring that the processor used only just meets their requirements.

2.3 The Problems with Benchmarks

One of the major problems with benchmarks is that it usually comes up with a single figure of merit. Or in Douglas Adam's words, a single answer to 'life, the universe and everything'. Benchmarks tend to measure the upper limit of machine performance in a single application area. This often is not a true reflection of the machine performance envelope. Various applications will generate different mixes of instructions and will place different requirements on the machine I/O subsystems. The result is that a MIPS, SpecInt or some other performance index may not have any relevance to how well a machine will perform in your application area. It has been shown [6] that effects such as congestion of the I/O channel and the interaction of the CPU core with its external environment may be key factors in determining machine performance for a specific application.

It should also be noted that the types of applications used by the different communities differs widely and a benchmark developed by one community may not necessarily be useful to another community.

Embedded applications generally have the advantage that the nature of the application is well-known beforehand, and will not greatly vary over the lifetime of the product.

2.4 The Myths of MIPS, MFLOPS and MOPS

These terms are all popular with CPU manufacturers, although their meaning is not always clear. Nor are the contexts in which they are quoted always fair.

The term MIPS stands for Million Instructions Per Second, and appears to have its origins in the VAX MIPS used by version two of the Dhrystone benchmarks of Dr Weicker, who was with Siemens AG at the time [7]. The general use of the term is used to define the theoretical maximum instruction rate any given machine can achieve. This, then, includes any *instruction* and not an *operation*.

An instruction is something that would be fetched from memory, be decoded and then executed. An operation not only includes instructions, but also other support activities that might be running in hardware in the background. As such DMA transfers would be counted as an operation, while a data move initiated by the core (from a fetched and decoded program

word) would be counted as a instruction. MIPS also defines the maximum instruction rate without taking into account whether the I/O subsystem can support the data moves mandated by that instruction.

By way of example – a machine with dual execution units running at a 100 MHz instruction rate would have a MIPS rating of 200 MIPS. In a real-world application the I/O subsystem may only be able to sustain 50 MIPS (depending on the machine architecture, of course).

The question that can be asked is where did the other 150 MIPS go to? It went into stalled cycles, of course, while the computer was waiting for data. These stalled cycles can be regarded as NOP's, which in turn can be regarded as instructions, which have to be counted! Our use (and that of the rest of the sane world) disregards these stalled cycles when counting MIPS.

The MIPS rating, therefore, only takes into account the capabilities of the machine execution unit(s), but does not consider the machine architecture as a whole.

The MFLOPS rating stands for Million Floating-point Operations Per Second, and refers to the ability of a CPU to do floating-point math. Its first use appears to be by the Livermore Loops benchmark of the 1970's [8]. This rating is especially relevant in the DSP community, where the ability of the machine to sustain floating-point intensive math may supersede any other requirement. This rating was mandated by the fact that the early RISC (and other) architectures needed several instruction cycles to complete floating-point math.

When the first DSP's arrived on the scene with their dedicated floating-point hardware, some way was needed to differentiate a machine's ability to execute instructions from its ability to do math.

By way of example, the Analog Devices ADSP21160 SHARC has dual cores, each with three dedicated maths units (ALU, multiplier and barrel shifter – a relative common DSP configuration). The two cores work in a SIMD (Single Instruction Multiple Data) fashion, which is executing the same instructions on different data sets in the same instruction cycle.

In theory in any given clock cycle each maths unit can do some calculation, for a total of six floating-point operations in every cycle. Because of the SIMD feature however, only three instructions per cycle are needed to do this. This would mean that at 100 MHz the ADSP21160 has a 300 MIPS and 600 MFLOPS rating. It is interesting to note that Analog Devices quote a maximum sustainable MFLOPS rating of 400 MFLOPS in their data sheets, thereby taking into account the capabilities (or restrictions) of the machine I/O subsystem.

The term MOPS appears to have originated with the Whetstone Benchmark [9]. It was also used by Texas Instruments to describe how much operations the TMS320C40 DSP processor could do in a single clock cycle. As such the term stands for Millions of Operations Per Second, and indicates, as described elsewhere in this chapter, how many operations of any kind (not only instructions, but also transfers such as that initiated by the DMA controller) the machine could do in a single cycle. At the time the C40 was the first DSP processor that designed with parallel features such COM ports (an extended version of the Transputer Link Ports) on the silicon, and some way was needed to describe this.

2.5 Dhrystone and Whetstone

The Dhrystone and Whetstone set of benchmarks are similar in intention to the SpecInt and SpecF benchmarks, with Dhrystone being the systems programming and Whetstone the floating-point variants.

Whetstone was the first of these benchmarks and was intended to test the floating-point math capability of the machines of the time. The original paper published was that of Curnow and Wichmann in 1967 [9]. According to a web reference [10], it was written by Harold Curnow of the British government procurement agency (CCTA) and was based on the work of Brian Wichmann of the National Physical Laboratory. Speed ratings initially were in Kilo Whetstones Per Second (KWIPS) but was updated in 1978 by Roy Longbottom to include MOPS and MFLOPS. Other versions for several different languages and applications (including Excel) can be found on the internet. The benchmark is not currently in general use.

Dhrystone was created by Dr Weicker in 1984 as a systems programming benchmark and was patterned on the Whetstone example. It is in essence a synthetic benchmark and uses the code distribution of applications of the time. It relies on the work of several other people and was released in several different languages, each version written to represent the typical code distribution of that language.

The benchmark contains three different types of statements. These are:

- Assignments such as a=b, a=constant, a = (expression of one operand - + - * / AND OR etc) , a= (expression of two operands) and a = (expression of three operands) and comparisons
- Control statements such as If..then, for loops, while loops and case statements
- Call statements such as procedure calls and function calls

In total 53% of statements are assignments (read integer math), 32% are control statements and 15% are function calls.

It is provided in a high-level language with the specific intent of testing not only the processor, but also the compiler, as the author feels that modern application will be written in a high-level language, and that the benchmark should represent this. The benchmark is not intended to test the complete machine architecture with the complete I/O subsystem included and the author recommends the use of other benchmarks to achieve this. The benchmark is also not intended to be used on its own, as the author recommends the use of other benchmarks as well. The issue of the whole benchmark fitting into the cache memory of a processor was understood and addressed in the first paper.

The benchmark was updated to version 2.1 [7] in 1989 to address mainly the issue of compiler optimisations, leaving out the dummy loops. The C version was also updated to reflect the development in C compilers. When the first version was released C was not yet widely used, and the compilers were in their first versions. By 1989 C was in widespread use, with the resulting new version of compilers and language extensions. As a matter fact, with version 2.1 it was recognized that C was the main system programming language used.

2.6 SpecInt and SpecF

In his second paper on the Dhrystone benchmark [7] Dr Weicker noted:

'If I were to write a new benchmark program, I wouldn't give it the name "Dhrystone" since this denotes the program published in [6]. However, I do recognise the need for a larger number of representative programs that can be used as benchmarks;...'

If Dr Weickner ever got his wish granted, SPEC is it.

The SPEC set of benchmarks is controlled by a consortium of workstation vendors, called the SPEC Open Systems Group or the Standard Performance Evaluation Corporation, as their documentation identifies them today. Dr Weickner left Siemens AG to join SPEC. The first version of SPEC appeared in the same year (1989) as his article on the updated version of Dhrystone was published.

SPEC is a frequently changing group of benchmarks [11], and comes in two flavours: SPECInt and SPECF. The SPECInt version is based around integer math, and the SPECF version around floating-point math.

The different versions of the SPEC benchmarks are known by their year of publication, The versions that currently exist are SPEC89, SPEC92, Spec95 and Spec2000. The SPEC2006 version is currently being worked on.

Each version uses a different set of programs, optimised to adequately exercise the machines of the time, to reflect real-world applications of workstations and to be as portable as possible. As the machines scale with time, it becomes possible for one or more of the programs to fit completely into the cache of the target system, and the runtimes decrease significantly. As the SPEC benchmarks calculate a ratio to a standard machine, the ratio becomes unacceptably high if the machines have scaled too much. This introduces big differences in SPECmarks with small changes in execution times. From there the need to change the benchmark suite. It

is probably the major point of criticism against the SPEC benchmark suite and is a problem other benchmarks such as LINPACK and HINT do not share.

The reference machines also change with time. The SPEC89 and SPEC92 benchmarks used the VAX-11/780 as reference, the SPEC95 version used the Sun SPARCstation 10 Model 40 workstation running at 40MHz and the SPEC2000 benchmarks use the Sun Ultra 5_10 workstation running at 300MHz as reference machines.

SPEC states that the different benchmark results are not comparable to each other, especially the SPEC95 and SPEC2000 results. They do, however, strongly encourage vendors to make both the benchmark results available to facilitate comparisons.

The benchmark apparently does not produce results that are exactly repeatable each time it is executed (with a variance of 7% being noted) [11]. It must be noted, however, that the source article only states a variance of 7% found on two identical workstations, but not the circumstances of the test, i.e. could the differences be accounted for by factors such as subtle differences in cache size, clock speed or workstation operating system version?

As with many other benchmarks, SPEC is also not decoupled from the effect of the compiler. [11],[12]. In fact, the SPEC2000 benchmarks list base and peak values, each reflecting different complier settings. At this point it should be observed that SPEC, LINPACK, HINT and other benchmarks all rely on the distribution of C source code, which is then compiled for the target architecture., The motivation, then, is that the benchmark does not test CPU performance per se, but rather the combination of CPU performance, target architecture capabilities and complier efficiency. There is, therefore, no indication of CPU potential, something an embedded designer, which is not averse to writing some assembly code, might be curious about.

It has also been shown that the static properties of the Spec95 set of programs differ substantially from the embedded code, and caution should thus be exercised when the Spec programs are used to benchmark embedded systems.[4]

So what does SPEC test?

From the SPEC website it is clear that two benchmarks are provided - CINT2000 and CFP2000. As the names would suggest, CINT2000 is the integer version, and CFP2000 is the floating-point version.

Quoting verbatim from the FAQ we find:

"*CINT2000 consists of 11 applications written in C and one in C++. These programs are:*

164.gzip	*Data compression utility*
175.vpr	*FPGA circuit placement and routing*
176.gcc	*C complier*
181.mcf	*Minimum cost network flow solver*
186.crafty	*Chess program*
197.parser	*Natural language processing*
252.eon	*Ray tracing*
253.perlbmk	*Perl*
254.gap	*Computational group theory*
255.vortex	*Object-orientated database*
256.bzip2	*Data compression utility*
300.twolf	*Place and route simulator*

CFP2000 contains 14 applications (six Fortran77, four Fortran90 and four C) that are used as benchmarks:

168.wupwise	*Quantum chromodynamics*
171.swim	*Shallow water modelling*
172.mgrid	*Multi-grid solver in 3D potential field*
173.applu	*Parabolic/elliptic partial differential equations*

177.mesa	*3D graphics library*
178.galgel	*Fluid dynamics: analysis of oscillatory instability*
179.art	*Neural network simulation: adaptive resonance theory*
183.equake	*Finite element simulation: earthquake modelling*
187.fracerec	*Computer vision: recognizes faces*
188.ammp	*Computational chemistry*
189.lucas	*Number theory: primality testing*
191.fma3d	*Finite-element crash simulation*
200.sixtrack	*Particle accelerator model*
301.apsi	*Solves problems regarding temperature, wind, distribution of pollutants"*

It can be seen that the list is impressive and indeed representative of industry applications. that makes SPEC, in my opinion, a good benchmark for desktop PC's, workstations and scientific parallel systems. It also explains why the benchmark is still popular after 15 years since its inception.

So what are the SPEC metrics?

The official SPEC2000 metrics are SPECint2000, SPECint_base2000, SPECint_rate2000 and SPECint_rate_base 2000 for the integer version, and SPECfp2000, SPECfp_base2000, SPECfp_rate2000 and SPECfp_rate_base2000 for the floating-point version.

Again from reference [11]:

The _base versions are compiled with conservative complier settings, and the other versions are compiled with aggressive compiler settings. The non-rate metrics are used to measure the ability of the computer to complete single tasks, and the rate metrics are used to measure the throughput of a machine carrying out a number of similar tasks (such as multiprocessor systems).

The SPEC ratings are basically the time it took the machine to complete the task, divided by the time it took the reference machine to complete the task. The rate metrics take into account the number of instances of the program that was run concurrently.

The source code for these benchmarks is not available for general circulation and must be obtained form SPEC directly.

2.7 Linpak

The LINPAK benchmark was originally written by Jack Dongarra of the Computer Science Department of the University of Tennessee [13].

The benchmark exercises mainly the ability of the machine under test to execute floating-point operations by solving sets of linear equations. Those readers still familiar with matrix algebra will remember Cramer's rule in which the matrix determinant as well as a partial determinant is calculated in order to solve the equations. Several other methods (such as L-D decomposition) for solving linear equations exist, and the author limits the method used only for matrix sizes of 100x100. This is done by distributing source code in FORTRAN. No limit is placed on the method of implementation for matrix sizes higher than 100x100. This allows vendors and users to try and find ways to improve the machine throughput through hand optimisations. This is also reflected in the method of reporting which states the MFLOPS throughput for the 100x100 and the 1000x1000 versions separately. Also included is the machine's theoretical throughput limit.

The benchmark difficulty level is scaled to the machine under test by varying the matrix size. In the test done by the author of LINPAK the problem sets varied between 100x100 matrix through 1000x1000 matrixes to 40000x40000 matrixes for the larger machines.

Mr Dongorra includes in his article some 50 pages of results for different machines and it is indeed interesting to note the differences between the MFLOPS achieved and the theoretical performance of the machine. The effect of hand optimisations is also clearly shown, and makes somewhat of a case to illustrate why developers of embedded systems are willing to spend significant manpower in hand optimisations of CPU code.

The benchmark is well-respected in the scientific community and its use is widespread throughout the community. The author could find no reference to its being used by the embedded community.

2.8 Touchstone

Touchstone is a very interesting small benchmark which was originally written by Mark Claypool of the Worcester Polytechnic Institute, Worcester, Massachusetts, in 1998. It is presented here by virtue of its ability to highlight the controversy surrounding computer benchmarks.

The original paper can be found at [14].

I quite like the quote at the beginning of the paper:

benchmark *v. trans* - to subject (a system) to a series of tests in order to obtain prearranged results not available on competitive systems. *S. Kelly-Bootle, The Devil's DP Dictionary*

This illustrates the somewhat tongue-in-cheek approach to benchmarks with surprising results.

The Touchstone benchmark is simple. All it consists of is a loop which increments a variable for a set time. The time was set to 200 seconds on a lightly loaded system through empirical measurements of the time it took to have repeatable results on a system. The increment can be either an integer or a floating-point value, thereby effectively creating two versions of the benchmark, TouchstoneInt and TouchstoneFP.

The benchmark was tested on nine different platforms, and the results were correlated with the performance benchmarks given by five other commercially available benchmark programs. These benchmarks were LINPAK, SPECInt, SPECfp, Quicksort and Gcc. In the

final analysis Touchstone correlated more than 98% with LINPAK, SPECInt and SPECfp. For Quicksort the correlation was 89% and 67% for GCC.

These results are rather profound. The reason is that Touchstone is small enough to fit into the cache of just about any processor; as a matter of fact, with the right compiler optimisations the count value should never have to leave the registers of the CPU core. Bigger benchmarks such as SPEC and LINPAK were written specifically to exercise not only the central processor, but also the entire machine, compiler included. This comparison asks serious questions on how successful those efforts were.

In defence of SPEC and LINPAK it should be noted that there is some uncertainty in the Touchstone results. The values listed in Table 2 of the article cannot possibly be the final count values after 200 seconds for all the processors. It would mean that an SGI Indigo 2 Extreme at 100 MHz could only count up to 5.5 million floating-point additions in 200 seconds. 5.5 million additions in 1 second would make more sense. LINPAK reports 15 MFLOPS for the same machine. The difference can be accounted for in the way MFLOPS are counted by LINPAK. Touchstone will not measure the ability of a processor to execute more than one floating point instruction per clock. LINPAK should do this.

Machine	TouchInt	TouchFp	Mflops	SPECint	SPECfp	Quicksort	Gcc
Crimson	9008886	5479664	16	58.3	63.4	0.84	-
Indigo	8802542	5367132	15.0	57.6	60.3	0.84	50.5
Sparc10	7044849	3445895	8.9	40.0	41.1	1.87	209.1
IPX	3506958	1702699	4.1	21.8	21.5	3.78	279.8
Sparc2	3580454	1709879	4.0	21.8	18.2	4.48	278.3
486	4096334	1230640	3.0	17.5	15.5	5.17	695.6
Iris	3721732	1331571	2.1	22.4	24.4	3.51	199.8
Sparc1	1737108	663700	1.4	9.5	7.5	9.31	530.7
386	638547	1730[1]	0.16	2.15	2.15	23.89	1590.1

Table 2: Experiment results. TouchInt and TouchFp are 200 second Touchstone values we measured when the increment variable is an integer variable or a floating point variable, respectively. Mflops are Millions of Floating Point Operations Per Second, as recorded by LINPACK. SPECint and SPECfp are integer and floating point intensive benchmarks recorded by the SPEC benchmark suite. Quicksort and Gcc are the run times in seconds for our Quicksort and Gcc experiments.

Figure 1 : Table 2 from [13]

The source code for this benchmark is rather trivial and can be found in [13].

2.9 Other Unix-type Benchmarks

Several other Unix-type benchmarks exist. Most were written by the scientific community for parallel type systems. These benchmarks include PARKBENCH, PERFECT and the NAS parallel benchmarking suite. They are not so different from the systems discussed here that they warrant individual investigation and are therefore not discussed in this document.

2.10 HINT

The HINT benchmark was initiated by John L Gustafson and Quinn Q Snell of the Ames Laboratory of the Department of Energy in the United States.

It is a synthetic benchmark and the name is an acronym for Hardware INTegration. It is the successor of the SLALOM benchmark, by the same author.

As its name implies, the benchmark attempts to test the interaction of the machine with its hardware environment, a noble goal. The benchmark successfully achieves this by using interval subdivision to calculate upper and lower bounds of sections of the mathematical function (1-x)/(1+x) (x is varied between 0 and 1). In essence the ranges for x and y are divided by two. It is then calculated which of the blocks thus defined completely fit below the (1-x)/(1+x) curve, and which are above the curve. These blocks are then stored. Each block is subdivided again, and the process is then repeated again and again. Each iteration then results in the improvement of the quality of the answer of where the bounds of the curve lie. With each iteration the number of stored blocks increases and thus memory is systematically filled. As memory use cascades from cache to main memory to external memory, the rate at which the quality of the answer is improved changes to reflect the mechanics of the memory from which the data is retrieved. The result is then that the impact of several factors relating to the machine and its environment is visible on a graph.

These factors are:

* Mathematical performance relating primarily to division and addition
* Memory access timing
* Machine instruction rate.

As with other benchmarks HINT is not decoupled from its software environment and factors such as memory allocated to the benchmark by the operating system and the efficiency of the compiler may skew results.

The benchmark metric is called QUIPS, or QUality Improvement Per Second. The references [15], [16], and [17] contain several such graphs. One such graph is depicted below.

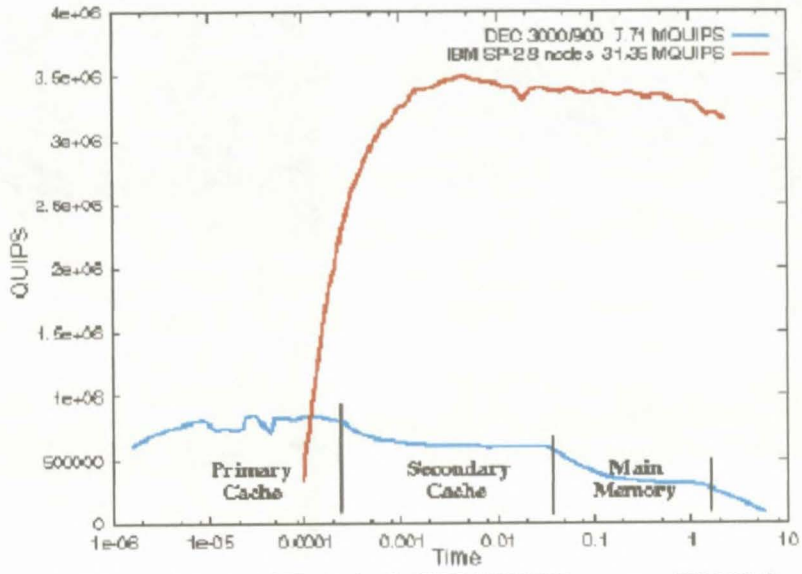

Figure 1: Example HINT graph of a DEC 3000/900 versus an IBM SP-2

Figure 2: A Sample HINT Graph

The benchmark is unique in the sense that it was the only benchmark the author could find that is able to give a performance curve over a range of operating conditions. Its main disadvantage is that it is a completely synthetic benchmark and contains no information, or even pointers' to information on its relevance to real-world applications. As such it also only tests a very small subset of the machine's instruction set, and even that is delivered to the mercy of the compiler. As a type of touchstone, however it is excellent as it is able to compare anything from a abacus to a supercomputer on a linear scale.

2.11 The Work of the EEBMC

The EEBMC is an acronym for the Embedded Microprocessor Benchmarking Consortium. It is a consortium of some 58 commercial companies (as of October 2004). These companies

include embedded microprocessor manufacturers such as Analog Devices, Intel, Toshiba and Texas Instruments.

The stated goal of the EEMBC is to provide a means for manufacturers of embedded processors to 'develop meaningful performance benchmarks for the hardware and software used in embedded systems' [18]

All hardware and software benchmarks have to be certified by the EEBMC Certification Labs (ECL), a commercial venture created for that purpose.

Because of the commercial nature of EEMBC, ECL and the companies involved, the complete objectiveness of the EEMBC is somewhat suspect from an academic point of view. This is illustrated by papers such as [5], which, although interesting to read, do attempt to convince the reader that the EEMBC benchmark suite is the only game in town.

Much like the SPEC benchmarks, the EEMBC benchmarks consist of a number of sample applications divided over several application fields. While most of the applications are based on real code, some such as the Cache "Buster" algorithm are completely synthetic.

For each applications field a performance mark is assigned. For each benchmark the number of iteration per second, code size and data size is stored. From this an overall score for that field is calculated. Other than stating that the mark is the geometric mean of all the performance results of the benchmarks in that field, no further information is given on the manner in which the benchmark score is derived.

The benchmarks included in version 2 are:

Automotive / Industrial (Automarks)

- Angle to Time conversion
- Basic Integer and Floating Point
- Bit Manipulation

- Cache "Buster"
- CAN Remote Data Request
- Fast Fourier Transform
- Finite Impulse Response Filter (FIR)
- Inverse Discrete Cosine Transform (iDCT)
- Inverse Fast Fourier Transform (iFFT)
- Infinite Impulse Response Filter (IIR)
- Matrix Arithmetic
- Pointer Chasing
- Pulse Width Modulation
- Road Speed Calculation
- Table Lookup and Interpolation
- Tooth to Spark

Consumer

- High Pass Grey-Scale Filter
- JPEG
- RGB to CMYK Conversion
- RGB to YIQ Conversion

GrinderBench for the Java 2 Micro Edition (J2ME) Platform

- Chess
- Cryptography
- kXML
- ParallelBench
- PNG Decoding

- Regular Expression

Networking

- Packet Flow
- OSPF
- Route Lookup

Office Automation

- Dithering
- Image Rotation
- Text Processing

Telecom

- Autocorrelation
- Bit Allocation
- Convolutional Encoder
- Fast Fourier Transform
- Viterbi Decoder

3. A New Approach

3.1 Summary of the Current State of Technology

It is clear from the preceding chapter that there are many conflicting requirements when defining benchmarks. The exact nature of the benchmark does of course depend on who is benchmarking. From my point of view the requirements of the different communities are as follows:

3.1.1. The Scientific Community

This domain is inhabited mainly by academics and the parallel computer communities. They are also mainly responsible for the plethora of synthetic and hybrid benchmarks. Their need appears to be for benchmarks that are scientifically answerable and repeatable on many machines and environments. They like to include the compiler in their benchmark methodology. The benchmark also needs to be scalable to accommodate the benchmarking of highly parallel computers. Many of the benchmark papers therefore treat the scalability of that benchmark as an important requirement. Most of the benchmarks discussed above fall in this category.

3.1.2 Desktop Applications

This community likes to benchmark real applications on desktop PC's. Benchmarks of this type is found everywhere in the commercial domain and many commercial computer magazines benchmark various systems using applications of their choice. An example would be the EDN magazine benchmarks. These benchmarks are not so much of scientific interest as it is of interest to a PC user wanting to decide where to spend his hard-earned cash.

3.1.3 The Embedded World

Very little work appears to have been done by the scientific community to benchmark embedded systems. The only attempt of which the author is aware of is the semi-commercial EEMBC. Their benchmark suite can be classified as a hybrid method, employing samples of real code together with some fully synthetic benchmarks.

3.2 CPU Benchmark Basics

From the previous chapter it is clear that almost all benchmark methods and programs measure the speed of execution of some standard application on various machines. As mentioned previously, and at the risk of repeating myself, while such benchmark methods may provide useful information in relating the speed of one processor to that of the other, it may be insufficient to help the designer decide which processor to use for his application. In order to do this we need to find some way in which we can relate the application to the processor under consideration. In order to obtain the necessary information a new benchmark method is needed.

3.2.1 Reinventing the benchmark

As with all such reinventions, we will start from scratch – that is consider the basic factors that make up CPU performance. In this regard our previous studies into benchmarking methods may help us.

We will start the process by further investigating the two most common technical aspects of computer benchmarking as uncovered in chapter two, namely that of CPU core performance and that of I/O performance. The investigation must, however, be done carefully, as both of these aspects of computer performance are complex and consist of several pieces of a puzzle, which, only when fitted together in the correct way, will provide a coherent picture of the whole.

3.2.2 CPU Performance

CPU performance in its simplest meaning refers to the ability of the CPU core to do work, where work is understood to mean the processing of instructions and data.

Various ways are used to measure work, of which the author find the *instructions retired* rate the most useful as it is an indication of how fast the CPU is processing instructions.

The *instructions retired rate* is, of course, the rate at which instructions are completed within the core.

The rate at which instructions are retired, of course, depends on other factors within the CPU. These factors, as perceived by the author, are:

- The rate at which instructions can be supplied to the core
- The rate at which the operands (or data, if you will) for the currently executing instructions can be supplied to the core
- The rate at which processed results can be moved away from the core
- The rate at which the core can process the instructions (i.e. the number of clock cycles needed to execute the instruction, if it is assumed that the instruction and its operands are available to the core)

It is clear that the first three items above are a function of the I/O subsystem and can thus be left for the paragraph dealing with the I/O subsystem. The author therefore now hazards the following definition of CPU core performance:

CPU core performance *is the rate at which the CPU core can execute the mix of instructions demanded by the application if no I/O cycle penalties are incurred.*

It is therefore stated that the rate at which a CPU can retire instructions is the same as the rate at which the core can execute those instructions if, and only if, no I/O penalties are incurred. With I/O penalties is meant those clock cycles the CPU core stalls while waiting for the moving of operands and instructions to the core, or processed results away from the core.

This is an important conclusion as it implies that processor core performance can be made to be orthogonal from I/O performance and can thus be characterised separately from the I/O performance of the core.

Orthogonality is, of course, achieved on paper by assuming that no I/O penalties are incurred, and by then calculating the core performance under those ideal conditions. The I/O penalties are calculated by assuming the core introduces no delay and then calculating the cycles needed to transfer the data.

The next issue is that of which metric to use to measure the said processor core performance. The most tempting, and certainly the most widely used metric, is that of time, and then usually in terms of the one or the other version of instructions per second. By instructions per second is usually meant the number of instructions which can be retired every second while the CPU is executing the code which is used for benchmarking effort. But the time needed by the core to execute an instruction should be handled with care as it is a compounded measurement. By this is meant that the time the core needs to complete an instruction depends on the number of core clock cycles needed to execute instruction and the rate at which those core clock cycles occurs. Thus:

Core execution time = number of core cycles needed * core period

Since clock period is the inverse of the clock rate ($F_c = 1 / t_{clk}$) the equation can be rewritten as:

$$\text{Core execution time} = \frac{\text{number of core cycles needed}}{\text{Core clock rate}} \qquad \text{[EQ1]}$$

The author prefers to leave clock speed out of the measurement as doing so will allow a more general measure of CPU core performance over differing architectures and clock speeds. It will as such provide a measure of a particular architecture's efficiency on the user's application.

The metric which will henceforth be used is that of the number of clock cycles required to complete the instructions, and which then is a measure of CPU efficiency as opposed to that of the CPU instruction rate as measured in time. The conversion between rate and efficiency

can easily be achieved through the use of equation 1, with the unit of measure then being that of instructions per second.

The method used to measure the core performance of the CPU should be general enough to allow for the differing schemes used by CPU manufacturers to improve the performance of the CPU core. The methods used fall into two major categories, namely:

- The addition of instruction pipelines to improve the instruction throughput rate, and
- Allowing multiple instructions per cycle to achieve superscalar performance.

3.2.3 Pipelining and other core performance enhancing features

Pipelining of the sequencer is used to enhance the performance of the program sequencer by allowing the sequencer to start work on the next instruction before the current instruction has been completed. This can be done as the normal flow of operation of a CPU sequencer consists of several steps which are orthogonal to each other. Most sequencers have the following order:

- Instruction fetch
- Instruction decode
- Instruction execution
- Save results

By fetching the next instruction while the previous instruction is being decoded etc., time is saved and an average throughput of one instruction per clock cycle (or more in the case of superscalar processors) can be achieved. The downside of pipelines is encountered, however, during the execution phase when it is discovered that the instruction being executed produces a result that causes program control to branch to another location in memory and that the instructions that are currently being fetched and decoded are therefore invalid. This is called a pipeline miss and core stalls are introduced while the correct instructions are being fetched. The number of cycles thus wasted is called a pipeline miss penalty.

This introduces an interesting dynamic in our CPU core performance measurement as it presents a case where the instruction execution time is dependant on the program flow. A condition which may be difficult to model since Turing has so aptly demonstrated that the

prediction of program flow is a NP complex problem and can thus not be accurately modelled.

3.2.4 I/O Performance

I/O performance is arguably the most important aspect of CPU design today. In short, it is the ability of the CPU to provide both instructions and data to the core in a timely fashion. If it cannot do this, core stalls will result.

Much of the modern CPU performance enhancing features has to do with improving I/O performance. Although modern CPU's can run at clock speeds of 3 GHz and higher, the I/O subsystem is nowhere near duplicating that. The reason for the discrepancy is that the further the distance the bits have to travel, the more difficult the electrical interconnection becomes due to effects such as parasitic capacitance. The other caveat is that fast memory is expensive. Hence the current CPU strategy of placing small fast memories on wide I/O buses close to the processor with larger slower ones being placed further away, and also the concept of cache memories storing the most used instructions and data close to the CPU and lesser-used instructions and data further away.

3.2.5 A Simple CPU model.

Given the preceding paragraphs, a simple CPU model can be compiled as shown in Figure 2 below.

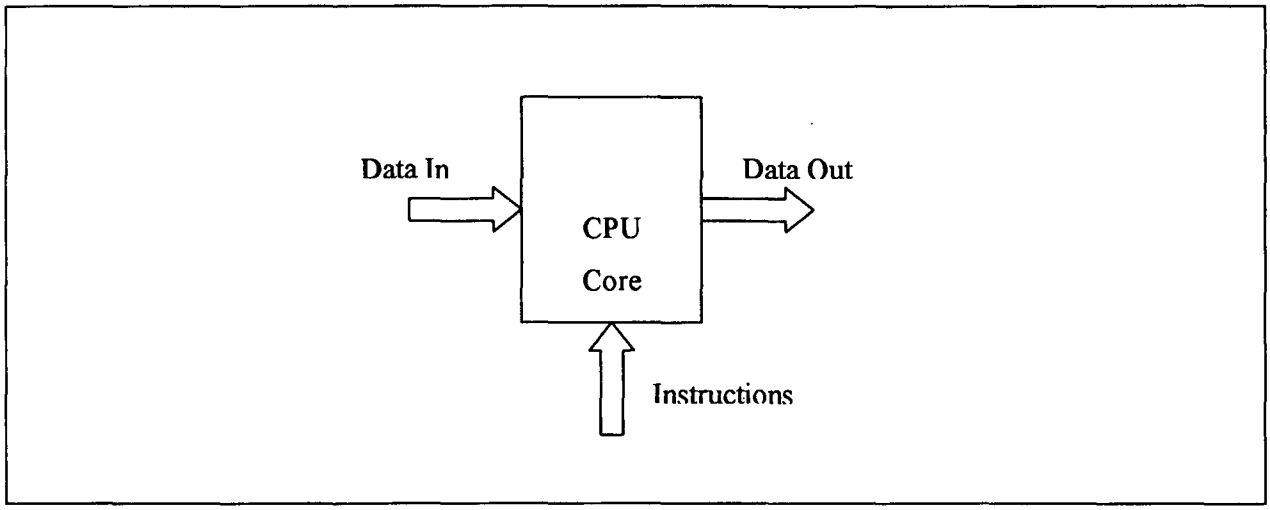

Figure 3: CPU Model

Please note that this model regards cache memory as being part of the I/O subsystem, and it is thus not shown here.

From the above drawing the following can be derived:

Every instruction executing in the core has a native I/O requirement inherent to it. In order for no stalls to occur, the I/O subsystem should be able to move all those bits in the time it takes the instruction to execute. Thus, if the demand on the I/O subsystem exceeds its capacity, core stalls will result. The latency (or number of care stalls) introduced by the I/O subsystem for a specific instruction can be calculated from the following formula:

$$Latency = \left\lceil \frac{IO\_bits\_Required}{IO\_bits\_Available} \right\rceil \qquad \text{[EQ2]}$$

Since no fractions of clock cycles are possible the equation can be rewritten to be:

$$Latency = Ceiling\left[\frac{IO\_bits\_Required}{IO\_bits\_Available} \right]$$

$$= Ceiling\left[\frac{(Input\_IO\_bits + Output\_IO\_Bits)}{IO\_Available} \right] \qquad \text{[EQ3]}$$

where ceiling is the Excel function which rounds up to the next whole integer.

It is also clear that the I/O latency will only be a factor if the actual instruction executes in less time than is required to move the bits. Thus:

Execution cycles = Max [IO latency, Instruction latency] [EQ4]

3.2.6 Graphic representation

Since any modern CPU has more that one instruction, and the I/O demand and the number of cycles required differs between instructions, some way of representing this data is required. The author chose the method of 3D surface plots where the X-axis is the data locality (i.e. where the bits are being sourced from), the Y-axis is the I/O requirement and the Z-axis is the latency. Such a graph for just the I/O latency is depicted in figure 3 below.

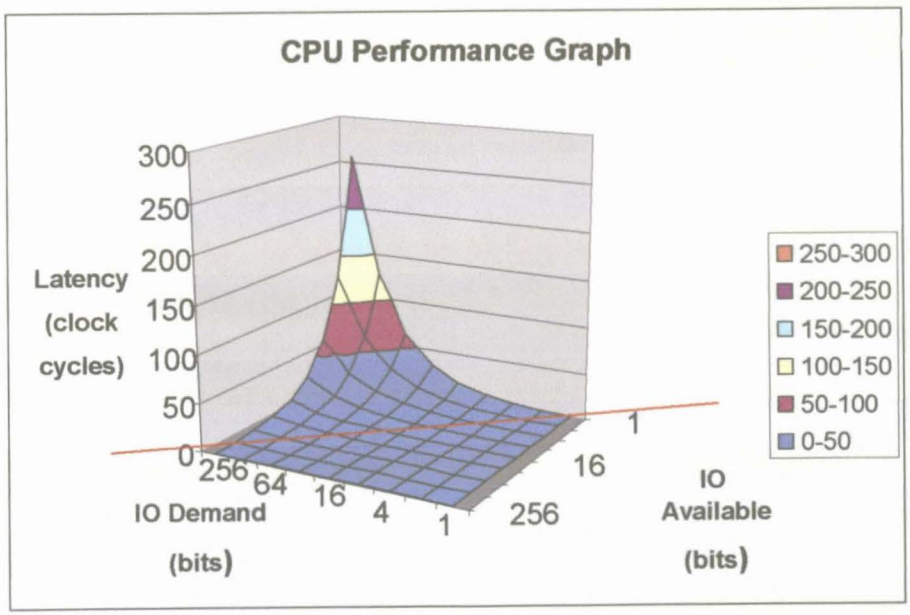

Figure 4: A Generic CPU Performance Graph

When the core latency is factored in, the result is a 3D graph depicting a performance profile for the CPU over a variety of instructions and I/O needs. The performance of an application on that CPU can then be predicted by analysing the application and determining the part of the CPU performance curve on which it will execute, and by then factoring in the clock speed

of the CPU. The efficiency at which a specific architecture will execute the application can also be determined by leaving out the CPU clock speed and just measuring the number of cycles required.

From the above graph it can also be seen that I/O latency only becomes an issue if I/O demand exceeds I/O availability. The answer is not unexpected, but is nevertheless important, as it is very likely that most processor memory will be located in areas where I/O demand may exceed I/O availability.

3.3 CPU Profiles for Selected Processors

We will now model and graph a selection of processors. As we will have to test the CPU profiles against real code, the selection of processors is restricted to those for which compilers are readily available. These processors are then:

- The ADSP21160N, A DSP processor from Analog Devices with SIMD capability
- The TS101 Tigersharc superscalar DSP processors from Analog Devices with VLIW technology
- The TMS320C67 superscalar DSP processor from Texas Instruments with VLIW technology.

The latter two processors are chosen not only because the compilers for them are available to the author but also because the basic CPU technology (superscalar DSP with VLIW technology) is the same, however with somewhat differing I/O architectures. The result is that the same applications can be tested on both processors, and the effect of the different I/O architectures can then be explored and compared.

Three graphs were compiled for each processor. These were then:

- The I/O Latency graph. This graph depicts the expected latency that will be introduced if the relevant number of operands is to be sourced from the memory in question.
- I/O Latency with cache misses. This graph depicts the cost of having to source instructions with data over the normal I/O channels, such as will happen if a instruction cache miss occurs.

- Number of Instructions Sustained depicts the number of cycles a given instruction (with its associated operands) can be maintained from the relevant memory location without data reuse. This is an indicator of the size of the data set that can be stored in that memory space.

In the compiling of the graphs the following observations and assumptions were made:

1. In Equation 3 the number of output I/O bits was assumed to be on average half of the input bit requirement. This comes from the fact that on average only one operand is stored back to memory after a mathematical operation. This is of course true for normal additions and subtractions. For multiplies the output operand width is the same as the input operand width (the multiplication of two 32 bit numbers produces a 64 bit answer), but the machine architecture allows only the transfer of a single operand wide answer back to internal memory. As such the answer is then normalised back to a 32 bit number (in the case of the processors at hand) and stored back to memory.

2. As every instruction in these machines complete in one clock cycle, the execution of an instruction never takes longer than the time needed to move the data required by the instruction, Equation 4 then becomes:

$$Latency = Max[Instruction\ latency,\ IO\ latency]$$
$$= IO\ Latency$$

Only the IO latency is therefore considered in the performance modelling and evaluation of these machines.

3. The graphs will assume the worst case scenario, i.e. that all data and instructions are sourced from the same locality. The graphs which explore the penalty of a cache miss will therefore assume that the instruction word required will be sourced from the same memory space as where the data is located, and that every machine cycle incurs a cache miss penalty.

The graphs were generated in Matlab and the source is attached as Appendix D.

3.3.1 The ADSP21160N Hammerhead

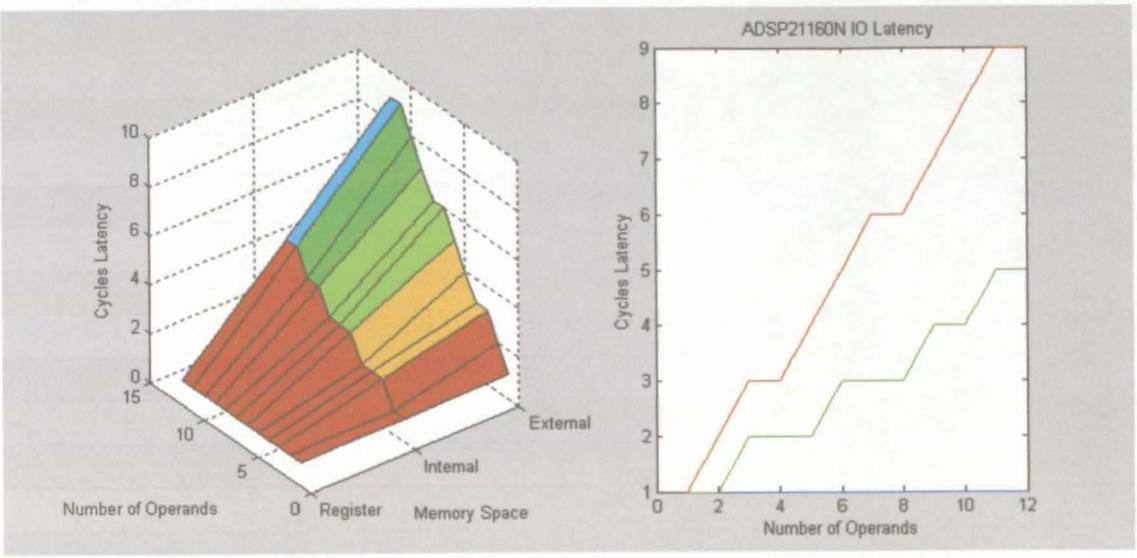

Figure 5: ADSP21160 I/O Latency

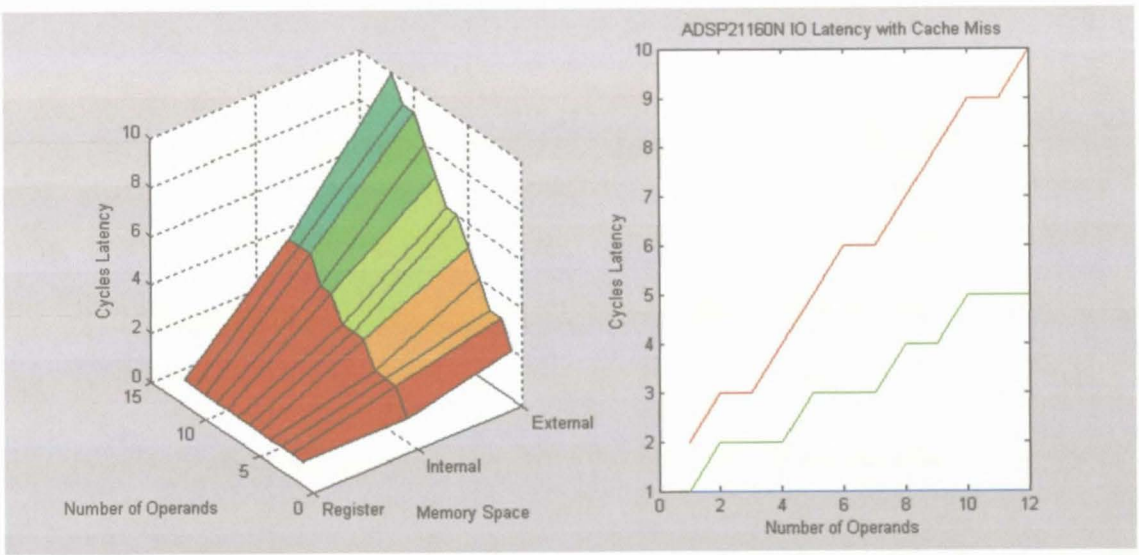

Figure 6: ADSP21160 I/O Latency with Cache Miss

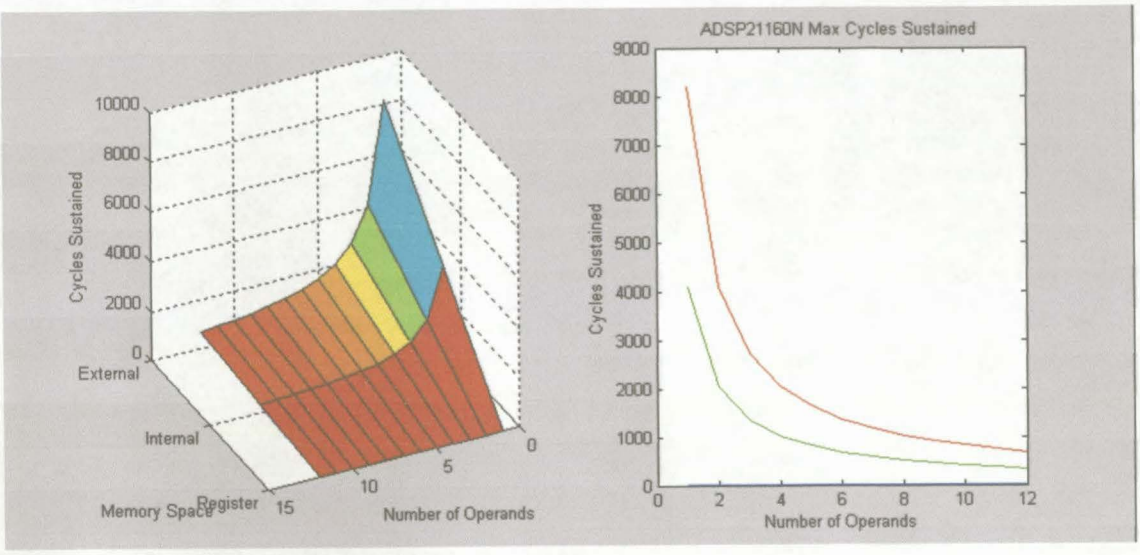

Figure 7: ADSP21160 Maximum Cycles Sustained

3.3.2 The TS101 TigerSHARC

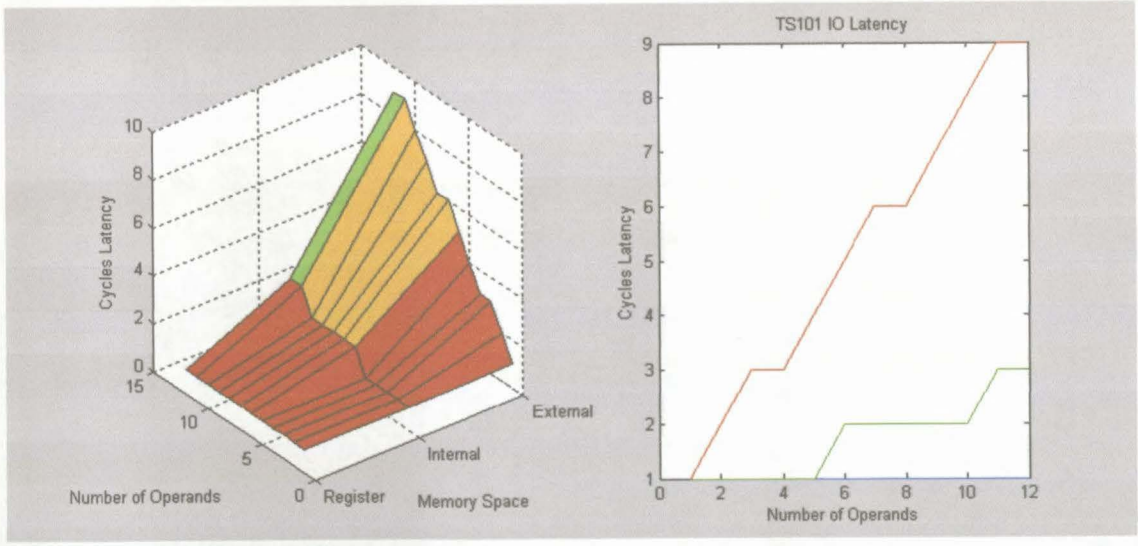

Figure 8: TS101 I/O Latency

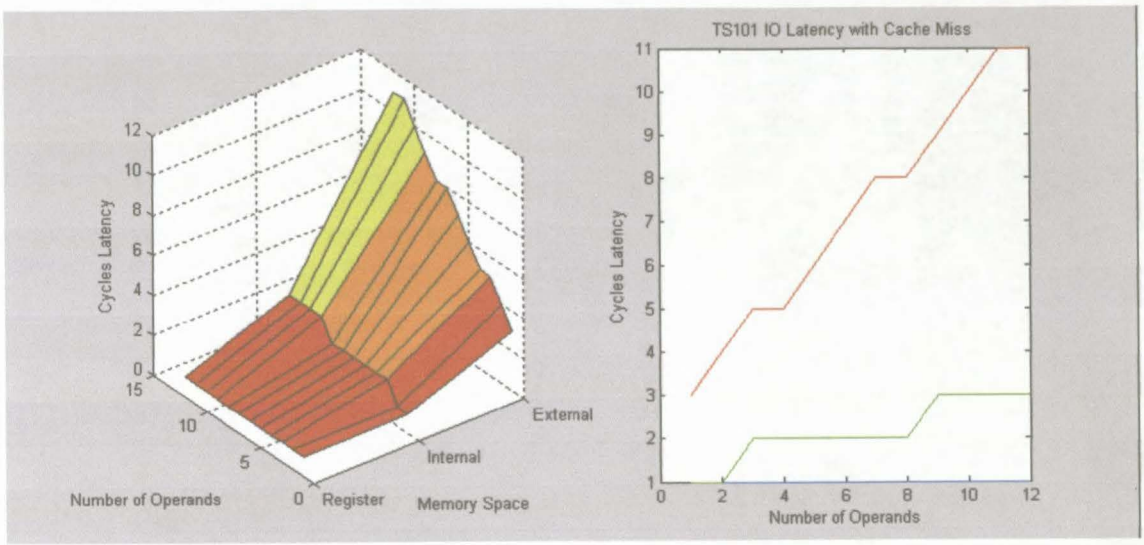

Figure 9: TS101 I/O Latency with Cache Miss

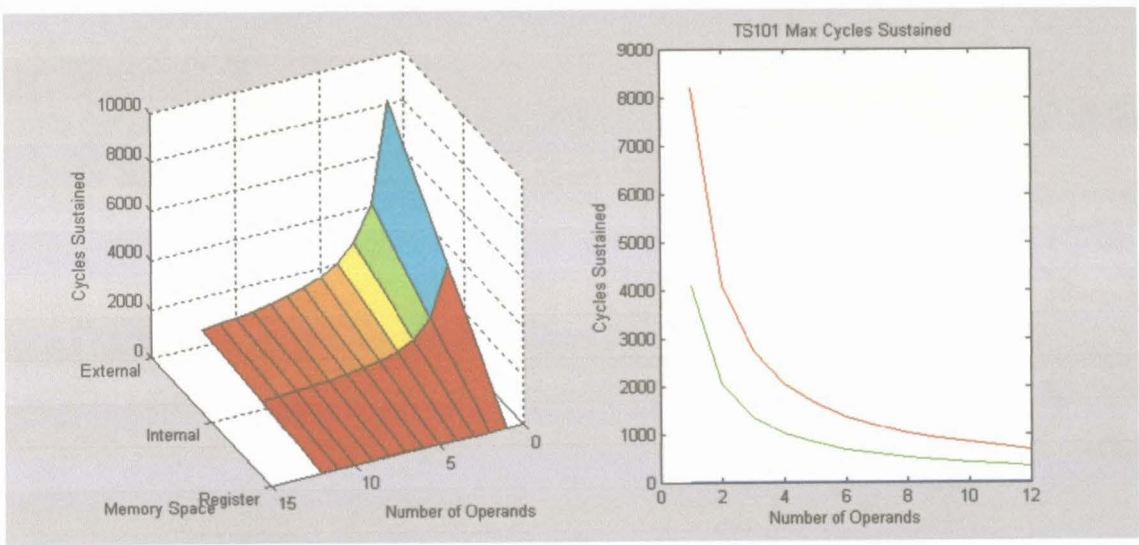

Figure 10: TS101 Maximum Sustained Cycles

3.3.3 The TMS320C67

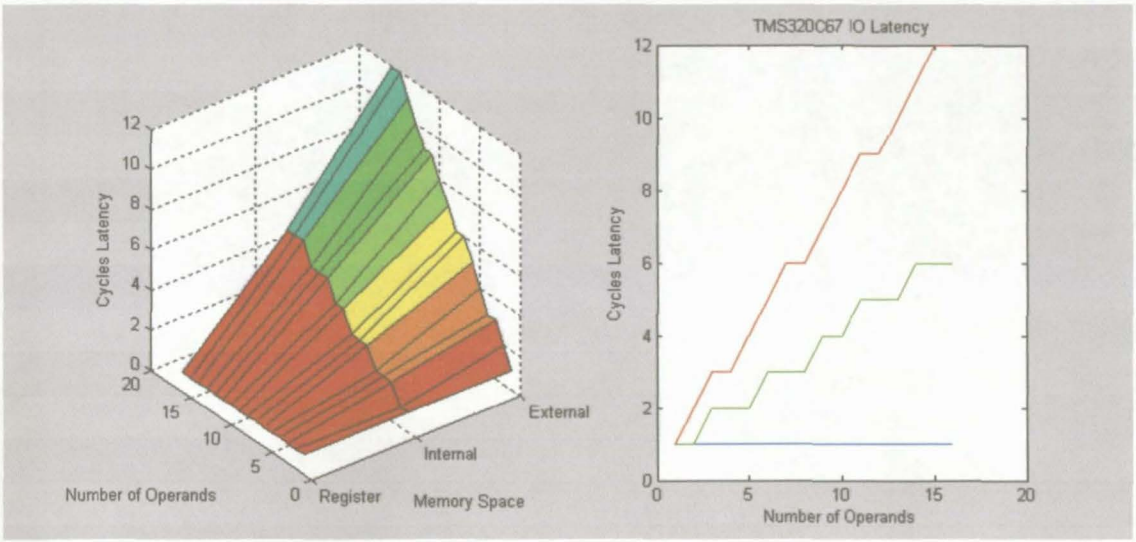

Figure 11: TMS320C67 I/O Latency

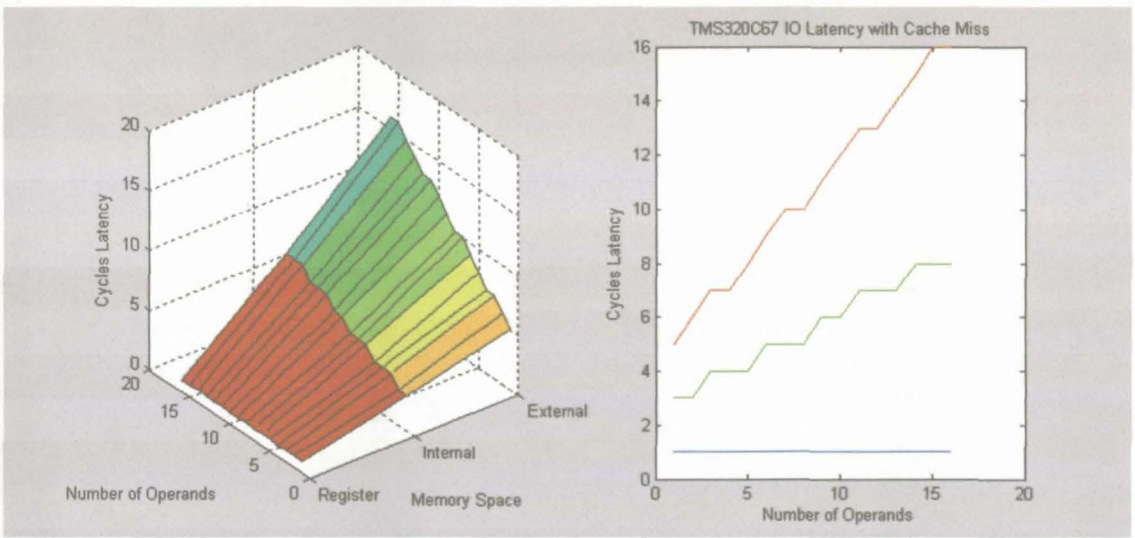

Figure 12: TMS320C67 I/O Latency with Cache Miss

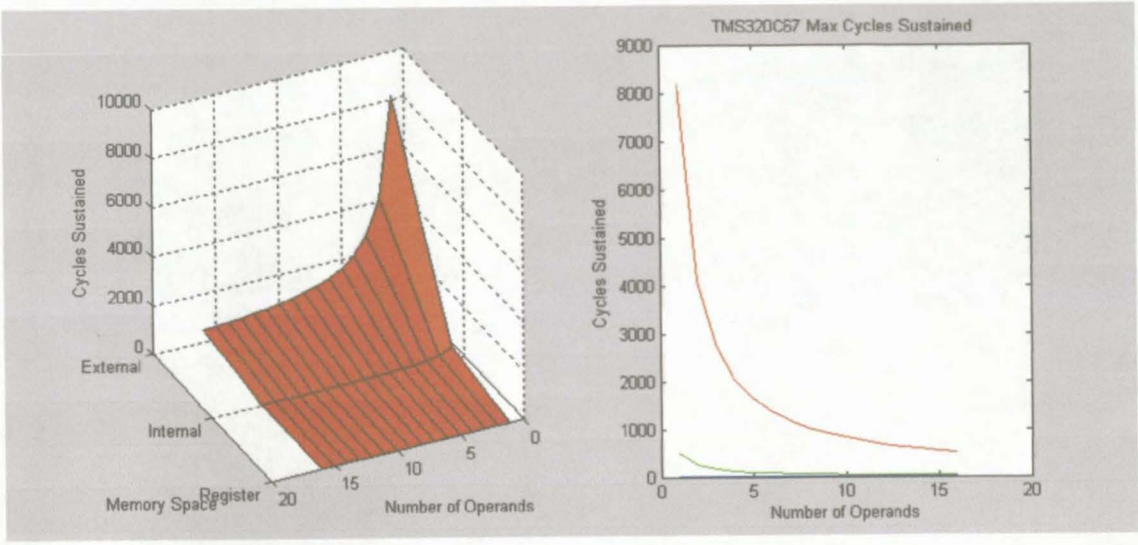

Figure 13: TMS320C67 Maximum Sustained Cycles

3.4 Observations and Conclusions

When compared to each other the differences in the ways the processors utilise their I/O bandwidth becomes quickly apparent.

The statistics are as follows:

	Internal Busses	External Bus	Instruction Word
ADSP21160N	2 x 64 bits	1 x 64 bits	48 bits
TS101	2 x 128 bits	1 x 64 bits	128 bits
TMS320C67	2 x 64 bits	1 x 64 bits	256 bits

Table 1: Processor Bandwidths

Firstly it is apparent that all three processors have the same external bandwidth. Thus when Figures 4, 7 and 10 are compared the external memory latencies are the same for all three processors. But when Figures 5, 8 and 11 are compared there are big differences with the ADSP21160N and the TS101 both being similar with substantial lower I/O latencies than the TMS320C67. The implication is (and it holds true for all cases) are that as much as possible of the program code need to be stored within the processor's internal memory. Especially for the VLIW machines. When the above conclusion is viewed in conjunction with Figures 6, 9 and 12 it is also clear that the TMS320C67 has relatively little internal memory. This creates a problem as the viable size of a data set that can be shared in internal memory with program code is not large. The implication is that the TMS320C67 will have a very busy external bus

(and associated long I/O latencies) for data set sizes of well below a 1000 points. The Analog Devices processors would then be more suited for high bandwidth I/O applications such as image processing applications, whereas the TMS320C67 would be happier in its intended Telecoms applications where the data rate is lower and each data word are more intensively processed.

The next observation concerns the use of internal memory. The TS101 by far outstrips the other processors when both data and instructions are located within the chip The ADSP21160 fares somewhat better and the TMS320C67 the worst of the lot.

The following table compares the processors in terms of the maximum I/O latency with a cache miss and the maximum usable data set size which can be accommodated in internal memory

	Maximum I/O Latency	Usable Data Set Size
ADSP21160N	5	2000
TS101	3	2000
TMS320C67	9	<500

Table 2: Internal Memory I/O Latency and Usable Data Set Size

The TS101 seems to be especially efficient when multi-operand instructions are used (such as will be the case when the processor's superscalar features are utilized). Once again the ADSP21160N places second with the TMS320C67 third.

The third observation is that when operating on register-to-register instructions, all the processors experience no bottlenecks.

The last observation is that both Analog Devices processors experience no additional I/O latencies when instructions with 0 or one operand is executed, even with cache miss circumstances. This indicates that both processors should do well in control type applications where large portions of code can be placed in external memory, even if such code contains many branches.

4. Application Benchmarking

4.1 Introduction

In this chapter we will demonstrate how to use the CPU profiles as defined in Chapter 3 to obtain estimates of the performance of selected algorithms on the CPU architectures of chapter 3.

4.2 First steps

Since chapter 3 provides us with a framework describing the performance envelope of the machine under test over a range of instruction types and memory types, we should now have enough information to decide how fast a given algorithm will run on that machine. In other words, it should now be possible to characterize (or map) the application and then match it to the performance envelope of chapter 3 in such a way that conclusions can be made as to the number of cycles the application will need to complete on the target machine.

There are a number of requirements this application mapping process should meet in order for it to be useful. Let me describe it in words first, and then make a list of steps to follow.

The process should be simple and intuitive. It must not require the coding of software for the target processor; the application algorithm should be enough. It must be relatively accurate, where the term 'relatively accurate' is understood to imply that the accumulated error should be a acceptable fraction of the spare capacity the CPU is intended to have. In most case errors in the range of 10 to 20% should be in order.

However, before we can begin we first need some information on the application we will be characterizing. We need to determine:

The algorithm written out to the last detail, and
The amount of storage it will require, i.e. the size of its data set.

In order to be able to answer these questions, the designer should have done his homework first. This implies that most of the system design work up to algorithm design should already be complete. This further implies that the designer should first design his algorithm and then the hardware and software platforms on which he will implement his algorithm. The designer should also have a good knowledge of the characteristics of the dataset his application will be using since that knowledge, when viewed in conjunction with the hardware characteristics of the machine under test, determines where on the hardware platform the data can reside. And as Chapter 3 so aptly demonstrated, the locality of data is in all likelihood the biggest single determining factor of the speed at which the application can execute.

The conclusion is that, although this benchmark will help a designer to determine the performance of his algorithm on a CPU before he has either built the hardware or written a single line of code for it on that machine, it does not free him of the obligation of knowing what he wants to do by first developing and testing the algorithm before benchmarking it, nor of having some basic knowledge of the hardware architecture of the machine he will be benchmarking on.

4.3 The basic process

The only way to determine the number of cycles an algorithm will require to execute on a machine is to determine the type and frequency instructions the machine will use. The next step will then be to determine the number of operands each instruction will use and from there how the I/O subsystem of the machine will suit the algorithm. Suitability is, of course, defined in the number of cycles required to move the operands. We can then determine if, for that algorithm, the machine is I/O or processing limited and hence the number of cycles that will be required to execute it.

The list that was promised in chapter 4.2 above will then look something like the following:

- Break up the algorithm into a list of discrete steps to be followed.
- Determine the type of operations each step will require and
- Match that list of operations to the capabilities of the CPU under test and optimize for concurrency (by this is meant grouping operations together that can execute in a single

instruction cycle. For this, of course, a working knowledge of the CPU instruction set is required – read that Instruction Set Reference manual!)

- From the graphs of chapter 3 determine the number of cycles required for each step.

4.3 The limits of benchmarking

The above steps are all fine, but there are some catches. From the above description it is evident that in order for us to benchmark any algorithm, we must know where the program will start, where it will end and how many steps were followed in between. This may or may not be a (rather knotty) problem. The reason for this has to do with the problem of computability.

4.3.1. Computability

The issue of computability touches on the work done by many others in the field of computer science, and dates back to the days of Turing and his quest to prove the *entscheidungs (or decision) problem* through the use of his Turing machines.

So what is the entscheidungs problem, and what has it or Turing Machines to do with us? The answer lies not so much in the decision problem itself nor in Godel's answer to that problem in the form of his Completeness and Incompleteness theorems nor even in Turing's proving of the Incompleteness theorem through his Turing Machines, but rather in the applications of the term *algorithm* or *table of instructions* as defined by Turing for his machines. You see, Turing machines were the theoretical definition of what we today know as computers. And in the days before computers actually existed, mathematicians theorized on what these theoretical machines would be able to compute.

At this point a short definition of computability is in order: An algorithm is computable if it can be translated into a series of instructions, which can be executed on a computer. (Note that this is the author's own definition of computability.)

Of all the issues raised in the research of computability, there are only two that directly concern us:

In order for us to be able to benchmark an algorithm it should not only be computable, but it should also be computable in linear time.

This means that we cannot benchmark algorithms, which follows a complicated decision tree and where the results of the decision are dependent on external data that is unknown to us. We have to be able to predict when and how often a branch decision will be taken.

Examples of such algorithms are:

Functions that converge on an answer such as the numeric calculation of a square root and where the number of iterations needed are data dependant. In some cases the calculation may not converge at all.

Algorithms where the program flow is significantly affected by real-time data taken from the environment. An example might be the control algorithm of a missile flight computer where decisions are made based on environmental factors such as wind speed and the dynamics of the target.

In short, any algorithm that is unpredictable in its nature will be difficult to benchmark.

All is not however lost for such algorithms, as these algorithms can usually be broken up into smaller parts, each of which may be predictable. It is possible to determine how long one iteration of the square root program will need to complete, or in the case of the missile controller, how many cycles the state variable update portion of the control algorithm will use.

4.3.2 Code style

Another factor that can influence code execution dramatically is the code style. Poor coding style almost always leads to a *loss* of performance. Since we want to benchmark before we have translated the algorithm into software, we should be aware of the consequences of various coding styles.

Firstly, some definitions (these are my own definitions and is based on the general use of the terms, but adapted to suit the problem at hand):

Coding: The coding of software is that process which allows an algorithm to be translated into software.

Coding Style: The coding style of the software is indicative of the method used to code the software. As such it represents the sum of a few factors such as whether or not the code was written in a high-level language, the optimizations used and the individual preferences of the programmer. Each piece of software will have a coding style in which the sum of all these factors is reflected.

Coding style concerns us insofar as there are good coding styles and poor coding styles. The coding style determines how *efficiently* the algorithm is translated into software. As such the coding style introduces a *derating* factor into our benchmark results. It should be noted at this point that the benchmark method as defined in chapter 3 uses assembly level code and generally returns a result of the best that architecture is capable of. *In essence this benchmark method states that all things considered some method of programming (read coding style) may be found which will allow the architecture at hand to implement the given algorithm in such a way that it will execute in the number of cycles indicated.*

So what are the major code styles?

This is of course not an easy question. Code styles are as individual as the programmer or compiler (and compilers are written by programmers!) that uses them. There are, however, major contributors to code style.

The author classifies code styles by the type of optimizations they use. Optimizations, of course, determine how well the underlying computer architecture is utilized. There are two major classes of optimizations:

(and once again the author makes his own definitions)

4.3.2.1 Memory use optimizations

Memory use optimizations reflect directly on how well the localities of especially intermediate variables are chosen. The author's definition of intermediate variables are those data values that resulted from a previous calculation and will be reused in the core within a few cycles. This is especially relevant to the RISC architectures, as these architectures cannot do calculations on data directly in memory. The load-store methodology so often used by these machines dictates that the data must first be fetched from memory and stored in a register. After the calculation has been completed the result must then be stored back to memory. This, of course, introduces extra cycles, which negatively affect performance. The coding styles, however, will determine how well the load-store architecture is managed and can have a drastic effect on performance.

Consider the following example:

C source:

```
C = A + B * C;
D = C + E;
```

The compiler will generate assembly code in the following form (each line represents one instruction cycle):

```
Store machine state
Fetch A
Fetch B
Fetch C
Intermediate = B * C
C = intermediate + A
Store C
Restore machine state
Store machine state
Fetch C
Fetch E
```

D = C + E

Store D

Restore machine state

Some compilers have an optimization switch. In the case of many compilers this means *register optimization.* The code will then be as follows:

Store machine state

Fetch A

Fetch B

Fetch C

Intermediate = B * C

C = intermediate + A

Fetch E

D = C + E

Store C

Store D

Restore machine state

Note that in the latter case the value of C is retained in a register and is reused by the next calculation, thereby saving some cycles.

4.3.2.2 Memory optimisations.

This refers to the process of intelligently exploiting the memory architecture of a machine by placing the dataset in memory where it can use the machine I/O subsystem most efficiently. An example may be a strategy where the code and data is swapped into memory in such a way that while the algorithm is running no references to external memory is required. A further optimization will be to have a DMA process bring a new block of data into internal memory in the background while the primary process is running.

4.3.2.3 Computer architecture optimizations

These optimizations refer to how well the specialized functions of the CPU core architecture functions are used. Compilers very seldom use these optimizations, and programmers that write code for multiplatform compatibility deliberately avoid it. Programmers may also not have a sufficient grasp of the machine architecture to use these optimizations well. Sometimes the processor designers are also at fault for not creating instructions that will exploit their machine architecture well or by making the use of such instructions excessively complicated.

Optimizations in this class are superscalar features such as SIMD capability, dedicated next address calculation hardware and zero overhead loop capability.

If we were to refer to the previous example, and was to implement it on the ADSP21160, the code written by a programmer may look something like this:

Store machine state
Fetch C, Fetch B
Fetch C, Intermediate = B * C
C = intermediate + A
Store C, Fetch E
D = C + E
Store D
Restore machine state

Note that the above example makes use of both register optimization and the superscalar features of the architecture and uses half the number of cycles of the first iteration.

Further enhancements are possible by using the SIMD features of the architecture. If SIMD was to be enabled, and providing that the data set is orthogonal, the Y core can execute the same instructions as the X core, but on a different data set. This will result in two calculations being completed in the same number of cycles as the SISD calculation, thereby again halving the effective number of cycles needed to complete the calculation.

4.3.2.4 Code Style Definitions

The author now hazards the following code style definitions:

a. Compiler Code Style

This is the most inefficient code style of all and uses no optimizations. The penalty paid in the number of cycles needed for any given algorithm is usually in the order of 6 to 10 times slower. [4]

b. Optimized Compiler Code Style

This is Compiler Code Style with register optimizations. It may also include memory optimization.

c. Assembly Code Style

This can be the most efficient code style, as proved by the optimized libraries for super-computers. This style, if used efficiently, will use register optimizations, memory optimizations and architecture optimizations.

4.4 Selected benchmarks

The benchmarks that are presented here are by no means an exhaustive list. They were chosen because they are for one P complete (and thus numerically orientated), and because the number of cycles they use on processors can be verified by a third party. Examples will be the benchmarks manufacturers use to quote performance figures for their own machines, and benchmark programs provided by various sources.

The latter characteristic is important as it allows us to verify our predictions against the claims of the manufacturer or benchmark provider, and should they differ, the sources code can usually be readily obtained. Having the source code available will help us to determine if our benchmark process is at fault or if the third party is at fault.

The author generally expects the benchmark to be much more optimistic than the manufacturers code as experience has shown that software programmers in general do not know (or do not care to know) how to properly exploit a CPU's hardware features.

In such cases recommendations will be made as to how the library routines can be improved to enhance performance.

This sometimes apparent dichotomy is of concern to the author as the manufacturer often attempts to gain market leverage by quoting such benchmarks. The result is, of course, skewed user perceptions of what a set of hardware can or cannot do.

4.4.1 The FFT benchmark

The first of our benchmarks will be a 1024 point radix 2 complex Fast Fourier Transform. Come let us investigate and get to the core of the algorithm.

4.4.2.1 An introduction to the FFT algorithm

In the engineering world the need sometimes arises to determine the frequency content of an analog signal. In the analog signal domain a device called a spectrum analyzer is used to do this. This is, however, an expensive and complex piece of equipment and is not always suitable for the application at hand. If the signal can be digitized, however, we can apply a mathematical transformation to the signal to extract the frequency content from the signal. This transformation is known as the Discrete Fourier Transform (DFT). The DFT, however, is computationally intensive and it was for a long time not feasible to apply this transformation to real-time signals. This all changed with the advent of the Fast Fourier Transform.

The FFT is a specialized implementation of the Discrete Fourier Transform (DFT) and is optimized for execution in processors. The original description of the FFT can be found at [22]

The FFT attempts to reduce the number of calculations needed, and has at its foundation the observation that the DFT is broken down into a set of smaller DFT's. These sets can once again be broken down until the smallest set is arrived at, and in which only two-point DFT's are needed.

The breaking down process has as its result an algorithm which consists of several stages, and where each stage consists of N/2 two-point DFT's. The stages are the result of the breakdown process, and there are N/2 - 1 stages.

4.4.2.2 The mathematics of FFT's

The FFT samples are separated into a even set and a odd set. A DFT is then defined for each set. The results of this DFT is then again divided into two to make up 4 DFT's, etc. The complete FFT is then the sum of all these DFT's. The maths shows that each uneven set have to be multiplied by factor i.e. W_n^k. This value differs for each set and is referred to as the 'twiddle factors'.

The 2 point DFT at the base of the FFT is called a 'butterfly'.

The FFT consists of 3 stages ((N/2)-1) and each stage has 4 (N/2) butterflies. Each butterfly has two inputs, called the primary node and the dual node. For each stage the node spacing differs. This spacing is referred to as the 'dual node spacing'. Associated with each butterfly is a twiddle factor whose exponent depends on the group and the stage.

Whereas the input sequence is sequentially ordered, the output sequence is not. If the inputs are scrambled through a process called 'bit reversal' the twiddle factors are used in order, the output is produced in order and the butterfly is simplified.

If a generic butterfly calculation is considered, the maths takes on the following form:

The inputs are:

x0 + j y0 and x1 + j y1

The twiddle factor is C + j -S

The outputs are x0' + j y0' and x1' + j y1'

And

$x0' = x0 + [Cx1 - (-Sy1)] = x0 + C*x1 + S*y1$

$y0' = y0 + [Cy1 + (-Sx1)] = y0 + C*y1 - S*x1$

$x1' = x0 - [Cx1 - (-Sy1)] = x0 - C*x1 - S*y1$

$y1' = y0 - [Cy1 + (-Sx1)] = y0 - C*y1 + S*x1$

Direct observation shows that there are some products which need only be calculated once, and which can then be reused. These products are:

$a = C * x1,$

$b = S * y1$

$c = C * y1,$

$d = S * x1$

And the remaining calculations are then:

$x0 + (a + b)$

$y0 + (c - d)$

$x0 - (a + b)$

$y0 - (c - d)$

4.4.2.3 FFT Benchmarks

From the above paragraph it is apparent that the butterfly calculation at the heart of the FFT algorithm requires the following:

- 4 input data words and
- Generates 4 output data words
- 4 multiplies
- 4 additions
- 4 subtractions

Each of the processors can execute two multiplies and two additions in each clock cycle, and therefore needs 4 cycles on average for the calculations themselves, two cycles for 2

multiplies and two additions, and another two cycles to complete the subtractions. The first two cycles generate an I/O demand of 8 input operands and generates 4 output operands. From The I/O latency introduced by the I/O demand is depicted in Table 3 below.

CPU Type	Register	Internal Memory	External Memory
ADSP21160	1	3	6
TS101	1	2	6
TMS320C67	1	4	8

Table 3: FFT I/O Latency

Both Analog Devices processors are therefore *core limited(!)* for the FFT if the data can fit into internal memory (which it can), while the TMS320C67 has the same core and I/O latency. If the data is located in external memory all the processors are I/O *limited*. It should be noted that the TMS320 will from necessity operate at least partially from external memory as it does not contain enough internal storage space to accommodate the entire data asset.

The next table shows the number of cycles required to complete the FFT butterfly for each memory space.

CPU Type	Register	Internal Memory	External Memory
ADSP21160	4	4	6
TS101	4	4	6
TMS320C67	4	4	8

Table 4: Cycles Required for the FFT Butterfly Calculation

Since an FFT has N/2 butterflies and (Log N) – 1 stages, a 1024 point radix 2 complex FFT will have 512 butterflies in 9 stages, equating to 4608 butterfly calculations that need to be done.

The following table shows the number of cycles needed to complete the butterfly calculations.

	Register	Internal Memory	External Memory
ADSP21160	18432	18432	27648
TS101	18432	18432	27648
TMS320C67	18432	18432	36864

Table 5: FFT Number of Cycles

The results compare well for the Analog Devices processors, where the manufacturer claims 18288 cycles for the SIMD FFT [19]. Note that with some optimisations to the first few stages and twiddle factors of the algorithm itself, the number of cycles can be reduced to 10998 [20]. Texas Instruments claims FFT times that, when normalised back to cycles, show that the TMS320C67 need slightly more cycles to complete the FFT [21]. This is in line with expectations that the device will need somewhere between 18432 and 36864 cycles to complete the calculation.

The FFT source code for the ADSP21160N is listed in Appendix E.

4.5 Conclusions

The results presented here are preliminary. Much more work is required to verify them, but so far the results are surprisingly good. Most benchmark programs by their very nature are P complete (the users usually insist on an answer within an acceptable time frame), and since many of the benchmark results are verified over a number of processors, it may be especially beneficial to investigate some of them further. Further work is also required on how to apply the method described here to NP complete programs, possibly by breaking them down into discrete P complete sections.

5. Conclusion and Further Work

This project set out to determine whether a way could be found to benchmark any embedded algorithm on a number of processors without having to write actual code for those machines. As such the work was conducted in three phases.

In the first phase a comprehensive literature study was conducted to determine the extent and thinking in terms of current and researched benchmark methods. In the process more than 160 articles from differing sources such as Citeseer and the ACM digital library were consulted. It was discovered that embedded benchmarking is firmly settled in the commercial domain, and that not much academic research on the topic is being conducted. It was also discovered that most existing benchmark methods rely on the execution of a standard piece of software on different machines, and thereby tests only specific portions of the processor architecture. These benchmarking programs are also normally written in a high-level language and thus do not take into account the specialized features many embedded processors utilize. These benchmarks can therefore not be relied upon to accurately reflect the performance of a user application on a given machine.

In the second phase a generic method was developed which graphically portrays a performance curve for a given machine. The process was tested on three different architectures and interesting results and conclusions were obtained. A study was also undertaken on the scope of algorithms the benchmark would be applicable to, and it was found that in general the algorithms must fall within the P complete domain. The performance curve alone gave a good visual indication of the suitability of each machine to a range of applications. It was shown that from observation alone that SHARC DSP's will be good for high I/O demand applications, but that they will also do well in control type applications – exactly those applications the device is being used in the field for. It was also shown that the TMS320C67 would be better at lower I/O rate applications, but where each data point is intensively processed.

In the third phase a method was developed whereby an application can be characterized, and then from the work of chapter 3, be mapped onto a processor architecture. This was tested with the FFT algorithm for all three processors of chapter 3, and results were obtained that

very closely tracked with the manufacturers' own claims. It was also apparent that the TMS320C67 claims, although technically correct do not allow for the small internal memory size, and that the algorithm will as a consequence experience severe speed degradation.

Further work will focus on much more intensive testing of the applications being mapped, and then on a wider range of processors. The author has a special interest in the NP type of algorithms, the execution path of which may be difficult to predict.

The project achieved what it set out to do, and the author is especially pleased with the results obtained in the application benchmarking.

References

[1] COBHAM A., "The intrinsic computational difficulty of functions." *Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science*, 1964

[2] IMMERMAN, N., "Computability and Complexity", *The Stanford Encyclopedia of Philosophy (Fall 2004 Edition)*, Edward N. Zalta (ed.), http://plato.stanford.edu/archives/fall2004/entries/computability.

[3] WEICKER, R.D., "Dhrystone : A Synthetic Systems Programming Benchmark", *Commun. ACM*, 27(10):1013--1030, 1984. 15

[4] TALLA D., JOHN L.K., "Performance Evaluation and Benchmarking of Native Signal Processing.", *Proceedings of European Conference on Parallel Processing, Lecture Notes in Computer Science #1685*, 266-270, 1999

[5] WEISS A.R., "Dhrystone Benchmark, History Analysis "Scores" and Recommendations", *EEMBC White Paper*, 2002

[6] LENG T., ALI R., et al, "An empirical study of hyper-threading in high performance computing clusters", *Dell Computer Corp.*, U.S.A.

[7] WEICKER R.D., "Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules," *SIGPLAN Notices* 23, No. 8, 49 -- 62 (1988).

[8] MCMAHON F.H., "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range", Lawrence Livermore National Laboratory, Livermore, California, UCRL-53745, December 1986

[9] CURNOW H.J., WICHMANN B.A., "A Synthetic Benchmark", *Computer Journal* Vol 19, No 1 1976

[10] LONGBOTTOM R., "Classic Benchmarks", http://homepage.virgin.net/roy.longbottom/classic.htm .

[11] GUSTAFSON J.L., TODI R., "Conventional Benchmarks as a Sample of the Performance Spectrum", *The Journal of Supercomputing No 13, 321-342*, 1999

[12] WEICKER R.D., "On the use of SPEC benchmarks in Computer Architecture", Siemens Nixdorf, 1996

[13] DONGARRA J.J., "Performance of Various Computers Using Standard Linear Equations Software", http://www.netlib.org/benchweb/performance.ps.

[14] CLAYPOOL M., "Touchstone – A Lightweight Processor Benchmark", Worcester Polytechnic Institute, 1998

[15] "Hint Benchmarks of Commodity Systems", http://www.scl.ameslab.gov/Projects/old/ClusterCookbook/hint.html

[16] SNELL Q.C., GUSTAFSON J.L., "An Analytical Model of the HINT Performance Metric", Ameslab, 1996

[17] JL GUSTAFSON J.L., SNELL Q.C., "HINT: A New way to measure computer performance", *Proceedings of the 28th Annual Hawaii International Conference on Systems Sciences*, IEEE Computer Society Press, Vol 2, pages 292-401

[18] "About the EEMBC", http://www.eembc.com/About/index.asp

[19] "Implementing In-Place FFTs on SISD and SIMD SHARC Processors", *Engineer to Engineer Note*, EE-267, Analog Devices

[20] "cfft_simd", Analog Devices Visual DSP 3.5 library source code distribution.

Note: Many more sources (about 200 in all) were consulted, but as very few of them deal with the topic of embedded computer benchmarking, not many were relevant.

[21] "Texas Instruments TMS320C67xx benchmarks", http://focus.ti.com/dsp/docs/dspplatformscontentaut.tsp?sectionId=2&familyId=327&tabId=499

[22] BRIGHAM, E.O., "The Fast Fourier Transform", Englewood Cliffs, N.J., Prentice Hall, 1974.

Appendix A: MATLAB Source Code

```
% CPU Model Worksheet

clear;

% First the ADSP21160

% Data and Instruction widths
Native_Data_Width=32;
Instruction_Width=48;

%Internal and extrnal bus capasity
Internal_Bus_Width=64;
External_Bus_Width=64;
R_R_Width=18*Native_Data_Width;
IM_Width=2*Internal_Bus_Width;
EM_Width=1*External_Bus_Width;
Number_Cores=2;
Units_per_Core=3;

%Concurrency
Max_Conc_Instr=6;
Max_Operands_In=12;
Max_Operands_Out=6;

%Available memory
Num_Registers=32;
Internal_Mem=(4*1024*1024)/32;        % Change to reflect program and data
memory allocation
External_Mem=(8*1024*1024)/32;        % Change to suit application


for i=1:Max_Operands_In

% Calculate the I/O Latency

%IOLatency_21160(i,1)=0
IOLatency_21160(i,1)=ceil((i*Native_Data_Width +
(i/2)*Native_Data_Width)/R_R_Width);
IOLatency_21160(i,2)=ceil((i*Native_Data_Width +
(i/2)*Native_Data_Width)/IM_Width);
IOLatency_21160(i,3)=ceil((i*Native_Data_Width +
(i/2)*Native_Data_Width)/EM_Width);

%Calculate the cost of a cache miss

%CacheMiss_21160(i,1)=0;
CacheMiss_21160(i,1)=ceil((i*Native_Data_Width +
(i/2)*Native_Data_Width)/R_R_Width);
CacheMiss_21160(i,2)=ceil((i*Native_Data_Width + Instruction_Width +
(i/2)*Native_Data_Width)/IM_Width);
CacheMiss_21160(i,3)=ceil((i*Native_Data_Width + Instruction_Width +
(i/2)*Native_Data_Width)/EM_Width);

%Calculate the maximum time a operation can be sustained from that memory
%locality. Assumes calculations is done in place - no storage is needed for
%exit operands

%Sustained_21160(i,1)=0;
Sustained_21160(i,1)=ceil(Num_Registers/(i*Native_Data_Width ));
```

```
Sustained_21160(i,2)=ceil(Internal_Mem/(i*Native_Data_Width ));
Sustained_21160(i,3)=ceil(External_Mem/(i*Native_Data_Width ));


end

figure(1);
subplot(1,2,1)
h=surf(IOLatency_21160);
%i=1:Max_Operands_In
%plot(i,IOLatency_21160(i,2),i,IOLatency_21160(i,3),i,IOLatency_21160(i,4))
;
colormap hsv
%shading interp
%set(h,'EdgeColor','k')
%light('Position',[-2,2,20])
%lighting phong
%material([0.4,0.6,0.5,30])
%set(h,'FaceColor',[0.7 0.7 0],'BackFaceLighting','lit')
%view([30,25])
%axis([0 200 0 6 -8 8])
set(gca,'XTickLabel',{'Register';'Internal';'External'})
%set(gca,'CameraViewAngleMode','Manual')
xlabel('Memory Space')
ylabel('Number of Operands')
zlabel('Cycles Latency')
%xlabel('Number of Operands')
%ylabel('Cycles Latency')
%zlabel('Cycles Latency')

subplot(1,2,2)
i=1:Max_Operands_In
plot(i,IOLatency_21160(i,1),i,IOLatency_21160(i,2),i,IOLatency_21160(i,3));
xlabel('Number of Operands')
ylabel('Cycles Latency')
zlabel('Cycles Latency')

title('ADSP21160N I/O Latency')

figure(2)
subplot(1,2,1)
h=surf(CacheMiss_21160);
%i=1:Max_Operands_In
%plot(i,IOLatency_21160(i,2),i,IOLatency_21160(i,3),i,IOLatency_21160(i,4))
;
colormap hsv
%shading interp
%set(h,'EdgeColor','k')
%light('Position',[-2,2,20])
%lighting phong
%material([0.4,0.6,0.5,30])
%set(h,'FaceColor',[0.7 0.7 0],'BackFaceLighting','lit')
%view([30,25])
%axis([0 200 0 6 -8 8])
set(gca,'XTickLabel',{'Register';'Internal';'External'})
%set(gca,'CameraViewAngleMode','Manual')
xlabel('Memory Space')
ylabel('Number of Operands')
zlabel('Cycles Latency')
%xlabel('Number of Operands')
%ylabel('Cycles Latency')
%zlabel('Cycles Latency')

subplot(1,2,2)
```

```
i=1:Max_Operands_In
plot(i,CacheMiss_21160(i,1),i,CacheMiss_21160(i,2),i,CacheMiss_21160(i,3));
xlabel('Number of Operands')
ylabel('Cycles Latency')
zlabel('Cycles Latency')

title('ADSP21160N I/O Latency with Cache Miss')

figure(3)
subplot(1,2,1)
h=surf(Sustained_21160);
%i=1:Max_Operands_In
%plot(i,IOLatency_21160(i,2),i,IOLatency_21160(i,3),i,IOLatency_21160(i,4))
;
colormap hsv
%shading interp
%set(h,'EdgeColor','k')
%light('Position',[-2,2,20])
%lighting phong
%material([0.4,0.6,0.5,30])
%set(h,'FaceColor',[0.7 0.7 0],'BackFaceLighting','lit')
view([-115,30])
%axis([0 200 0 6 -8 8])
set(gca,'XTickLabel',{'Register';'Internal';'External'})
%set(gca,'CameraViewAngleMode','Manual')
xlabel('Memory Space')
ylabel('Number of Operands')
zlabel('Cycles Sustained')
%xlabel('Number of Operands')
%ylabel('Cycles Latency')
%zlabel('Cycles Latency')

subplot(1,2,2)
i=1:Max_Operands_In
plot(i,Sustained_21160(i,1),i,Sustained_21160(i,2),i,Sustained_21160(i,3));
xlabel('Number of Operands')
ylabel('Cycles Sustained')

title('ADSP21160N Max Cycles Sustained')

% Second the TS101

%clear;
% Data and Instruction widths
Native_Data_Width=32;
Instruction_Width=128;

%Internal and extrnal bus capasity
Internal_Bus_Width=128
External_Bus_Width=64

R_R_Width=18*Native_Data_Width;
IM_Width=2*Internal_Bus_Width;
EM_Width=1*External_Bus_Width;
Number_Cores=2;
Units_per_Core=3;

%Concurrency
Max_Conc_Instr=6;
Max_Operands_In=12;
Max_Operands_Out=6;

%Available memory
Num_Registers=32;
```

```
Internal_Mem=(4*1024*1024)/32;          % Change to reflect program and data
memory allocation
External_Mem=(8*1024*1024)/32;          % Change to suit application


for i=1:Max_Operands_In

% Calculate the I/O Latency

IOLatency_TS(i,1)=ceil((i*Native_Data_Width +
(i/2)*Native_Data_Width)/R_R_Width);
IOLatency_TS(i,2)=ceil((i*Native_Data_Width +
(i/2)*Native_Data_Width)/IM_Width);
IOLatency_TS(i,3)=ceil((i*Native_Data_Width +
(i/2)*Native_Data_Width)/EM_Width);

%Calculate the cost of a cache miss

%CacheMiss_TS(i,1)=0;
CacheMiss_TS(i,1)=ceil((i*Native_Data_Width +
(i/2)*Native_Data_Width)/R_R_Width);
CacheMiss_TS(i,2)=ceil((i*Native_Data_Width + Instruction_Width +
(i/2)*Native_Data_Width)/IM_Width);
CacheMiss_TS(i,3)=ceil((i*Native_Data_Width + Instruction_Width +
(i/2)*Native_Data_Width)/EM_Width);

%Calculate the maximum time a operation can be sustained from that memory
%locality. Assumes that calculations is done in place - no extra storage
%needed for exit operands

%Sustained_TS(i,1)=0;
Sustained_TS(i,1)=ceil(Num_Registers/(i*Native_Data_Width ));
Sustained_TS(i,2)=ceil(Internal_Mem/(i*Native_Data_Width ));
Sustained_TS(i,3)=ceil(External_Mem/(i*Native_Data_Width ));


end

figure(4);
subplot(1,2,1)
h=surf(IOLatency_TS);
%i=1:Max_Operands_In
%plot(i,IOLatency_21160(i,2),i,IOLatency_21160(i,3),i,IOLatency_21160(i,4))
;
colormap hsv
%shading interp
%set(h,'EdgeColor','k')
%light('Position',[-2,2,20])
%lighting phong
%material([0.4,0.6,0.5,30])
%set(h,'FaceColor',[0.7 0.7 0],'BackFaceLighting','lit')
%view([30,25])
%axis([0 200 0 6 -8 8])
set(gca,'XTickLabel',{'Register';'Internal';'External'})
%set(gca,'CameraViewAngleMode','Manual')
xlabel('Memory Space')
ylabel('Number of Operands')
zlabel('Cycles Latency')
%xlabel('Number of Operands')
%ylabel('Cycles Latency')
%zlabel('Cycles Latency')

subplot(1,2,2)
i=1:Max_Operands_In
```

```
plot(i,IOLatency_TS(i,1),i,IOLatency_TS(i,2),i,IOLatency_TS(i,3));
xlabel('Number of Operands')
ylabel('Cycles Latency')
zlabel('Cycles Latency')

title('TS101 I/O Latency')

figure(5)
subplot(1,2,1)
h=surf(CacheMiss_TS);
%i=1:Max_Operands_In
%plot(i,IOLatency_21160(i,2),i,IOLatency_21160(i,3),i,IOLatency_21160(i,4))
;
colormap hsv
%shading interp
%set(h,'EdgeColor','k')
%light('Position',[-2,2,20])
%lighting phong
%material([0.4,0.6,0.5,30])
%set(h,'FaceColor',[0.7 0.7 0],'BackFaceLighting','lit')
%view([30,25])
%axis([0 200 0 6 -8 8])
set(gca,'XTickLabel',{'Register';'Internal';'External'})
%set(gca,'CameraViewAngleMode','Manual')
xlabel('Memory Space')
ylabel('Number of Operands')
zlabel('Cycles Latency')
%xlabel('Number of Operands')
%ylabel('Cycles Latency')
%zlabel('Cycles Latency')

subplot(1,2,2)
i=1:Max_Operands_In
plot(i,CacheMiss_TS(i,1),i,CacheMiss_TS(i,2),i,CacheMiss_TS(i,3));
xlabel('Number of Operands')
ylabel('Cycles Latency')
zlabel('Cycles Latency')

title('TS101 I/O Latency with Cache Miss')

figure(6)
subplot(1,2,1)
h=surf(Sustained_TS);
%i=1:Max_Operands_In
%plot(i,IOLatency_21160(i,2),i,IOLatency_21160(i,3),i,IOLatency_21160(i,4))
;
colormap hsv
%shading interp
%set(h,'EdgeColor','k')
%light('Position',[-2,2,20])
%lighting phong
%material([0.4,0.6,0.5,30])
%set(h,'FaceColor',[0.7 0.7 0],'BackFaceLighting','lit')
view([-115,30])
%axis([0 200 0 6 -8 8])
set(gca,'XTickLabel',{'Register';'Internal';'External'})
%set(gca,'CameraViewAngleMode','Manual')
xlabel('Memory Space')
ylabel('Number of Operands')
zlabel('Cycles Sustained')
%xlabel('Number of Operands')
%ylabel('Cycles Latency')
%zlabel('Cycles Latency')
```

```
subplot(1,2,2)
i=1:Max_Operands_In
plot(i,Sustained_TS(i,1),i,Sustained_TS(i,2),i,Sustained_TS(i,3));
xlabel('Number of Operands')
ylabel('Cycles Sustained')

title('TS101 Max Cycles Sustained')

% Third the TMS320C67

%clear;
% Data and Instruction widths
Native_Data_Width=32;
Instruction_Width=256;

%Internal and extrnal bus capasity
Internal_Bus_Width=64
External_Bus_Width=64

R_R_Width=24*Native_Data_Width;
IM_Width=2*Internal_Bus_Width;
EM_Width=1*External_Bus_Width;
Number_Cores=2;
Units_per_Core=4;

%Concurrency
Max_Conc_Instr=8;
Max_Operands_In=16;
Max_Operands_Out=8;

%Available memory
Num_Registers=32;
Internal_Mem=(0.5*1024*1024)/32;        % Change to reflect program and data
memory allocation
External_Mem=(8*1024*1024)/32;        % Change to suit application


for i=1:Max_Operands_In

% Calculate the I/O Latency

%IOLatency_c67(i,1)=0
IOLatency_c67(i,1)=ceil((i*Native_Data_Width +
(i/2)*Native_Data_Width)/R_R_Width);
IOLatency_c67(i,2)=ceil((i*Native_Data_Width +
(i/2)*Native_Data_Width)/IM_Width);
IOLatency_c67(i,3)=ceil((i*Native_Data_Width +
(i/2)*Native_Data_Width)/EM_Width);

%Calculate the cost of a cache miss

%CacheMiss_c67(i,1)=0;
CacheMiss_c67(i,1)=ceil((i*Native_Data_Width +
(i/2)*Native_Data_Width)/R_R_Width);
CacheMiss_c67(i,2)=ceil((i*Native_Data_Width + Instruction_Width +
(i/2)*Native_Data_Width)/IM_Width);
CacheMiss_c67(i,3)=ceil((i*Native_Data_Width + Instruction_Width +
(i/2)*Native_Data_Width)/EM_Width);

%Calculate the maximum time a operation can be sustained from that memory
%locality. Assumes that calculations is done in place - no extra storage
%needed for exit operands

%Sustained_c67(i,1)=0;
```

```
Sustained_c67(i,1)=ceil(Num_Registers/(i*Native_Data_Width ));
Sustained_c67(i,2)=ceil(Internal_Mem/(i*Native_Data_Width ));
Sustained_c67(i,3)=ceil(External_Mem/(i*Native_Data_Width ));


end

figure(7);
subplot(1,2,1)
h=surf(IOLatency_c67);
%i=1:Max_Operands_In
%plot(i,IOLatency_21160(i,2),i,IOLatency_21160(i,3),i,IOLatency_21160(i,4))
;
colormap hsv
%shading interp
%set(h,'EdgeColor','k')
%light('Position',[-2,2,20])
%lighting phong
%material([0.4,0.6,0.5,30])
%set(h,'FaceColor',[0.7 0.7 0],'BackFaceLighting','lit')
%view([30,25])
%axis([0 200 0 6 -8 8])
set(gca,'XTickLabel',{'Register';'Internal';'External'})
%set(gca,'CameraViewAngleMode','Manual')
xlabel('Memory Space')
ylabel('Number of Operands')
zlabel('Cycles Latency')
%xlabel('Number of Operands')
%ylabel('Cycles Latency')
%zlabel('Cycles Latency')

subplot(1,2,2)
i=1:Max_Operands_In
plot(i,IOLatency_c67(i,1),i,IOLatency_c67(i,2),i,IOLatency_c67(i,3));
xlabel('Number of Operands')
ylabel('Cycles Latency')
zlabel('Cycles Latency')

title('TMS320C67 I/O Latency')

figure(8)
subplot(1,2,1)
h=surf(CacheMiss_c67);
%i=1:Max_Operands_In
%plot(i,IOLatency_21160(i,2),i,IOLatency_21160(i,3),i,IOLatency_21160(i,4))
;
colormap hsv
%shading interp
%set(h,'EdgeColor','k')
%light('Position',[-2,2,20])
%lighting phong
%material([0.4,0.6,0.5,30])
%set(h,'FaceColor',[0.7 0.7 0],'BackFaceLighting','lit')
%view([30,25])
%axis([0 200 0 6 -8 8])
set(gca,'XTickLabel',{'Register';'Internal';'External'})
%set(gca,'CameraViewAngleMode','Manual')
xlabel('Memory Space')
ylabel('Number of Operands')
zlabel('Cycles Latency')
%xlabel('Number of Operands')
%ylabel('Cycles Latency')
%zlabel('Cycles Latency')
```

```
subplot(1,2,2)
i=1:Max_Operands_In
plot(i,CacheMiss_c67(i,1),i,CacheMiss_c67(i,2),i,CacheMiss_c67(i,3));
xlabel('Number of Operands')
ylabel('Cycles Latency')
zlabel('Cycles Latency')

title('TMS320C67 IO Latency with Cache Miss')

figure(9)
subplot(1,2,1)
h=surf(Sustained_c67);
%i=1:Max_Operands_In
%plot(i,IOLatency_21160(i,2),i,IOLatency_21160(i,3),i,IOLatency_21160(i,4))
;
colormap hsv
%shading interp
%set(h,'EdgeColor','k')
%light('Position',[-2,2,20])
%lighting phong
%material([0.4,0.6,0.5,30])
%set(h,'FaceColor',[0.7 0.7 0],'BackFaceLighting','lit')
view([-115,30])
%axis([0 200 0 6 -8 8])
set(gca,'XTickLabel',{'Register';'Internal';'External'})
%set(gca,'CameraViewAngleMode','Manual')
xlabel('Memory Space')
ylabel('Number of Operands')
zlabel('Cycles Sustained')
%xlabel('Number of Operands')
%ylabel('Cycles Latency')
%zlabel('Cycles Latency')

subplot(1,2,2)
i=1:Max_Operands_In
plot(i,Sustained_c67(i,1),i,Sustained_c67(i,2),i,Sustained_c67(i,3));
xlabel('Number of Operands')
ylabel('Cycles Sustained')

title('TMS320C67 Max Cycles Sustained')
```

Appendix B: FFT Source Code

```
/*****************************************************************************
*
* Function:  CFFTx - Complex floating point FFT
*
* Prototype: complex_float *cfftx_simd( complex_float input[],
*                                       complex_float output[] );
*     where: x is the FFT length
*            such as cfft128_simd( input, output );
*
* Assumptions:
*            All arrays must start on even address boundaries
*
* (c) Copyright 2002 Analog Devices, Inc.  All rights reserved.
*     $Revision: 1.12 $
*
*****************************************************************************/

.swf_OFF; /* switch off ADSP-2116x Shadow Write Anomaly (SWFA) code */
          /* screening as this source is SWFA-safe                 */

#if   ( CFFT == 8 )
#define LENGTH    8
#define SIZE      3
#define NAME      _cfft8_simd
#define TWIDDLE   ___tc8
#elif ( CFFT == 16 )
#define LENGTH    16
#define SIZE      4
#define NAME      _cfft16_simd
#define TWIDDLE   ___tc16
#elif ( CFFT == 32 )
#define LENGTH    32
#define SIZE      5
#define NAME      _cfft32_simd
#define TWIDDLE   ___tc32
#elif ( CFFT == 64 )
#define LENGTH    64
#define SIZE      6
#define NAME      _cfft64_simd
#define TWIDDLE   ___tc64
#elif ( CFFT == 128 )
#define LENGTH    128
#define SIZE      7
#define NAME      _cfft128_simd
#define TWIDDLE   ___tc128
#elif ( CFFT == 256 )
#define LENGTH    256
#define SIZE      8
#define NAME      _cfft256_simd
#define TWIDDLE   ___tc256
#elif ( CFFT == 512 )
#define LENGTH    512
#define SIZE      9
#define NAME      _cfft512_simd
#define TWIDDLE   ___tc512
#elif ( CFFT == 1024 )
#define LENGTH    1024
#define SIZE      10
```

```
#define NAME       _cfft1024_simd
#define TWIDDLE    ___tc1024
#elif ( CFFT == 2048 )
#define LENGTH     2048
#define SIZE       11
#define NAME       _cfft2048_simd
#define TWIDDLE    ___tc2048
#elif ( CFFT == 4096 )
#define LENGTH     4096
#define SIZE       12
#define NAME       _cfft4096_simd
#define TWIDDLE    ___tc4096
#elif ( CFFT == 8192 )
#define LENGTH     8192
#define SIZE       13
#define NAME       _cfft8192_simd
#define TWIDDLE    ___tc8192
#elif ( CFFT == 16384 )
#define LENGTH     16384
#define SIZE       14
#define NAME       _cfft16384_simd
#define TWIDDLE    ___tc16384
#elif ( CFFT == 32768 )
#define LENGTH     32768
#define SIZE       15
#define NAME       _cfft32768_simd
#define TWIDDLE    ___tc32768
#elif ( CFFT == 65536 )
#define LENGTH     65536
#define SIZE       16
#define NAME       _cfft65536_simd
#define TWIDDLE    ___tc65536
#else
#error "You must define CFFT=xxx in the asm21k command, eg. -DCFFT=128"
#endif

#include <platform_include.h>
#include <asm_sprt.h>

.global NAME;
.segment /pm seg_pmco;

.extern ___bitrev_simd;
.extern TWIDDLE;

NAME:
   entry;
   puts = r3;
   puts = r5;
   puts = r6;
   puts = r7;
   puts = r9;
   puts = r10;
   puts = r11;
   puts = r13;

   r2 = i2;
   puts = r2;

   r2 = i3;
   puts = r2;

   r2 = m2;
   r0 = r0 - r0,
```

```
   puts = r2;

   r2 = m3;
   r9 = r0 + 1,
   puts = r2;

   s0 = r9;
   bit set mode1 PEYEN; nop;
   r0 = pass r0;
   bit tst astatx 0x00000001;          /* PEx TF set, PEy TF clear */
   bit clr mode1 PEYEN;
   r12 = LENGTH;                       /* le */
   lcntr = SIZE, do (pc,big) until lce;
      r0  = r12 + r12,                 /* le*2*2, and
*/
      m2  = r12;                       /* le*2 (complex interleaved data)
*/

      m3  = r0;
      r12 = ashift r12 by -1;          /* le = le / 2 */
      i2  = r4;                        /* &input[0].r */
      bit set mode1 PEYEN;
      i3  = i2;                        /* &input[0+le].r */
      modify( i3, m2 );                /* &input[0+le].r */
      f0 = dm( i2, m5 );               /* x[i].r */
      lcntr = r9, do (pc,stage1) until lce;  /* loop windex times */
         f1 = dm( i3, m5 );            /* x[i+le].r */
         f5 = f0 + f1, f10 = f0 - f1, f0 = dm( m3, i2 );
                                       /* x[i].r+x[i+le].r, x[i].r-
x[i+le].r */
         dm( i2, m3 ) = f5;            /* x[i].r = x[i].r + x[i+le].r */
stage1:  dm( i3, m3 ) = f10;           /* x[i+le].r = x[i].r - x[i+le].r
*/
      bit clr mode1 PEYEN;

      i4 = TWIDDLE;
      r0 = ashift r9 by 1;             /* windex *= 2 (interleaved
complex) */
      m4 = r0;

      r5 = r12 - 1,                    /* le - 1 */
      modify( i4, m4 );                /* wptr = &w[ windex ] */

      if eq jump (pc,revbits) (la);

      bit set mode1 PEYEN;
      r13 = 2;                         /* counter (1..n) *2 (for complex)
*/
      lcntr = r5, do (pc,stage2) until lce;
         r6  = r4 + r13, f7 = dm( i4, 0 );
         i2  = r6;                     /* &input[1].r */
         i3  = i2;
         modify( i3, m2 );             /* &input[i+le].r */
         f0 = dm( i2, m5 );            /* x[i].r */
         f1 = dm( i3, m5 );            /* x[i+le].r */
         f5 = f0 + f1, f10 = f0 - f1, s11 = dm( i4, m4 );
         if tf f11 = -f11;             /* set f11 only to -twid.i */
         lcntr = r9, do (pc,inner) until lce;
            f3 = f10 * f7,              dm( i2, m3 ) = f5;
            f6 = f10 * f11,             f0 = dm( i2, m5 );
            f6 <-> s3;
            f2 = f3 + f6,               f1 = dm( m3, i3 );
inner:      f5 = f0 + f1, f10 = f0 - f1, dm( i3, m3 ) = f2;
         r13 = r13 + 1;                /* increment counter */
```

```
stage2:  r13 = r13 + 1;                    /* *2 for interleaved complex */
      bit clr mode1 PEYEN;
big:  r9 = ashift r9 by 1;                 /* windex = windex * 2 */

revbits:
      r0 = r4;                             /* &input buffer */
      call (pc,___bitrev_simd) (db);
       r1 = pass r8,                       /* &output buffer,              */
       r3 = r8;                            /* and preserve &output buffer */
       r2 = LENGTH;                        /* length of buffer            */

done:
   r0  = pass r3,                          /* function result = &output */
   r3  = gets( 12 );
   r5  = gets( 11 );
   r6  = gets( 10 );
   r7  = gets( 9 );
   r9  = gets( 8 );
   r10 = gets( 7 );
   r11 = gets( 6 );
   r13 = gets( 5 );
   i2  = gets( 4 );
   i3  = gets( 3 );
   m2  = gets( 2 );
   m3  = gets( 1 );
   exit;

.swf_ON;
#if   ( CFFT == 8 )
._cfft8_simd.end:
#elif ( CFFT == 16 )
._cfft16_simd.end:
#elif ( CFFT == 32 )
._cfft32_simd.end:
#elif ( CFFT == 64 )
._cfft64_simd.end:
#elif ( CFFT == 128 )
._cfft128_simd.end:
#elif ( CFFT == 256 )
._cfft256_simd.end:
#elif ( CFFT == 512 )
._cfft512_simd.end:
#elif ( CFFT == 1024 )
._cfft1024_simd.end:
#elif ( CFFT == 2048 )
._cfft2048_simd.end:
#elif ( CFFT == 4096 )
._cfft4096_simd.end:
#elif ( CFFT == 8192 )
._cfft8192_simd.end:
#elif ( CFFT == 16384 )
._cfft16384_simd.end:
#elif ( CFFT == 32768 )
._cfft32768_simd.end:
#elif ( CFFT == 65536 )
._cfft65536_simd.end:
#endif
```